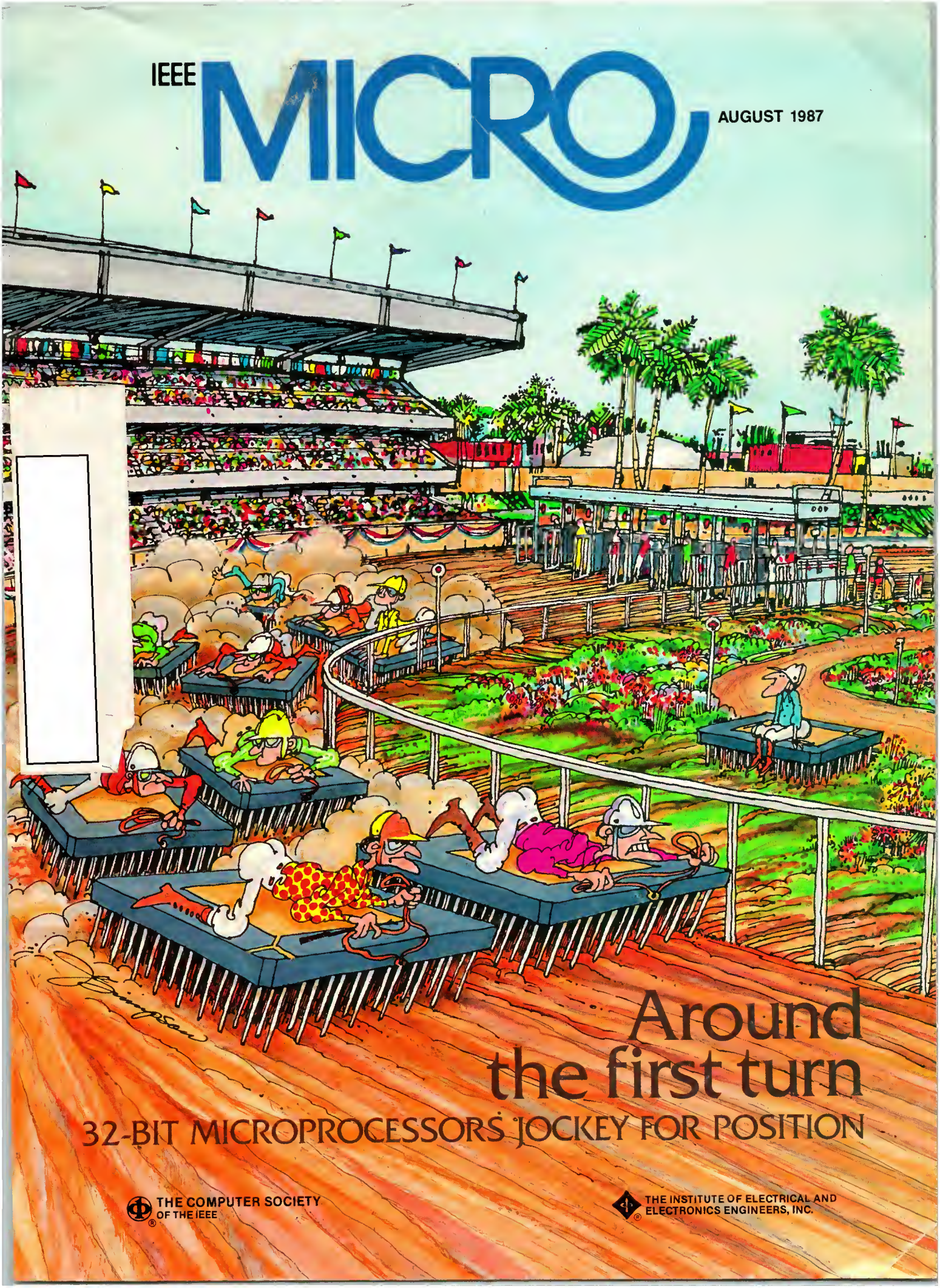
IEEE MICRO

AUGUST 1987



Around the first turn

32-BIT MICROPROCESSORS JOCKEY FOR POSITION

THE COMPUTER SOCIETY OF THE IEEE

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

On the cover
Picturing the serious competition surrounding 32-bit chips as a horse race turned out to be fun for cover designer Jay Simpson. *IEEE Micro* hopes you enjoy it too.

IEEE MICRO

Volume 7 Number 4 (ISSN 0272-1732) August 1987

Published by the Computer Society of the IEEE

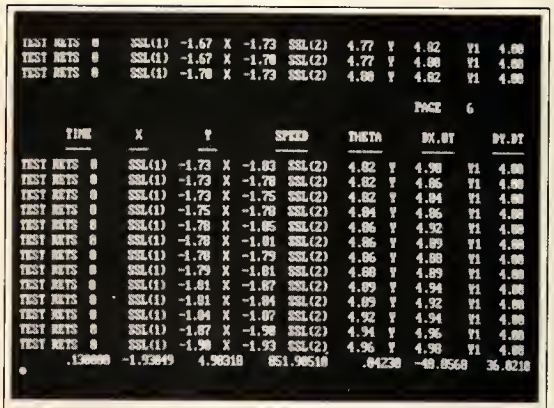
Departments

Feature Articles

Special Features

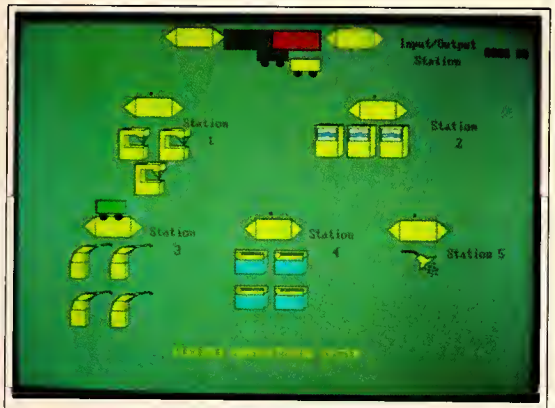
SIMULATION TEACHING BREAKTHROUGH

BEFORE



Difficult to understand results

AFTER



You see an animated picture of the system under study--any application

Now for universities
PC AT SIMSCRIPT II.5 with SIMANIMATION

New simulation teaching package--provides animated results

With PC SIMSCRIPT II.5 and new animation, simulation results are easy-to-understand--moving pictures, histograms, pie charts and plots.

Because simulation results are understood, you can concentrate on teaching simulation concepts.

Simulation animation simplified

SIMSCRIPT II.5 gives you a compact English-like language. The simulation program reads like a description of the system under study. Animation is easy.

Your students' model development, checkout, modification and enhancement are greatly simplified.

Many successful applications

SIMSCRIPT II.5® is a well established, standardized, and widely used language with proven software support.

Typical applications include: manufacturing, communications, logistics, transportation and military planning.

Low cost for universities

CACI recognizes the benefits of having its state-of-the-art systems used by the universities.

For that reason we make the PC SIMSCRIPT II.5 teaching package available to universities for only a low distribution charge.

The package includes training, support, complete documentation, and sample models. Everything you need to conveniently teach simulation analysis.

Act now-limited offer

This offer is limited to 1,500 universities, and 1,273 have already signed up. Don't be disappointed--call today.

For immediate information

Call Rick Crawford at (619) 457-9681. In the UK, call Steve Wombell on (01) 940-3606.

With PC SIMSCRIPT II.5 your students get results sooner and the results are better understood.

Rush information on the special simulation teaching package for universities only.

Everything you need to teach PC SIMSCRIPT II.5 with animation.

Also send information about:

- ☐ Mainframe SIMSCRIPT II.5.
- ☐ NETWORK II.5. Network analysis with no programming.
- ☐ SIMFACTORY. Factory planning with no programming.

Name

Organization

Address

City State Zip

Telephone

Computer Operating System

IEEE GRAPHICS

Return to:
CACI
3344 North Torrey Pines Court
La Jolla, California 92037
Or, better yet, call Rick Crawford at (619) 457-9681

In the UK
CACI
26 The Quadrant
Richmond, Surrey, England TW9 1DL
Call Steve Wombell on (01) 940-3606

SIMSCRIPT II.5, SIMANIMATION, and PC SIMSCRIPT II.5 are registered trademarks and service marks of CACI, INC.-FEDERAL

©1987 CACI, INC.-FEDERAL

IEEE MICRO

The Computer Society
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
(714) 821-8380

Editor-in-Chief:
James J. Farrell III, VLSI Technology Incorporated*

Associate Editor-in-Chief:
Joe Hootman, University of North Dakota

Editorial Board
Shmuel Ben-Yaakov, Ben Gurion University of the Negev
Dante Del Corso, Politecnico di Torino, Italy
John Crawford, Intel Corporation
Stephen A. Dyer, Kansas State University
K.-E. Grosspietsch, GMD, Germany
David B. Gustavson, Stanford Linear Accelerator Center
Victor K.L. Huang, AT&T Information Systems
Barry W. Johnson, University of Virginia
David K. Kahaner, National Bureau of Standards
G. Jack Lipovski, University of Texas
Kenneth Majithia, IBM Corporation
Richard Mateosian
Marlin H. Mickle, University of Pittsburgh
Varish Panigrahi, Digital Equipment Corporation
Ken Sakamura, University of Tokyo
Michael Smolin, Smolin & Associates
Richard H. Stern
Yoichi Yano, NEC Corporation

*Submit six copies of all articles and special-issue proposals to James J. Farrell III, 8375 South River Parkway, Tempe, AZ 85284; (602) 752-6222; Compmail + j.farrell.

Magazine Advisory Committee
Michael Evangelist (chair), Vishwani D. Agrawal, James J. Farrell III, Ted Lewis, David Pessel, True Seaborn, Bruce D. Shriver, John Staudhammer

Publications Board
J.T. Cain, (chair), Vishwani D. Agrawal, J. Richard Burke, Gerald L. Engel, Michael Evangelist, James J. Farrell III, Lansing Hatfield, Ronald G. Hoelzeman, Ted Lewis, Ming T. Liu, Ed Nahouraii, David Pessel, C.V. Ramamoorthy, Vincent D. Shen, Bruce D. Shriver, John Staudhammer, Steven L. Tanimoto

STAFF

Editor and Publisher: True Seaborn
Managing Editor: Marie English
Assistant Editor: Christine Miller
Assistant to the Publisher: Pat Paulsen
Advertising Director: Dawn Peck
Art Director: Jay Simpson
Design/Production: Tricia Hayden
Membership Manager: Christina Champion
Circulation Manager: Paul Zive
Advertising Coordinators: Heidi Rex, Marian Tibayan
Reader Service: Marian Tibayan

Moving?

☐Address changes:
Please notify us 4 weeks in advance

Name (Please Print)

New Address

City

State/Country Zip

ATTACH LABEL HERE

- This address change notice will apply to all IEEE publications to which you subscribe.
- List new address above.
- If you have a question about your subscription, place label here and clip this form to your letter.

Mail to:
IEEE Micro, Circulation Dept.,
10662 Los Vaqueros Cir.,
Los Alamitos, CA 90720-2578

Circulation: *IEEE Micro* (ISSN 0272-1732) is published by the Computer Society of the IEEE: IEEE Headquarters, 345 East 47th St., New York, NY 10017; Computer Society West Coast Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Annual subscription: $17 in addition to Computer Society or any other IEEE society member dues. $25 for members of other technical organizations. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Second class postage is paid at New York, NY, and at additional mailing offices.

Copyright and reprint permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US Copyright Law for private use of patrons: those post-1977 articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 29 Congress St., Salem, MA 01970. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint, or republication permission, write to Reprint, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All rights reserved. Copyright © 1987 by the Institute of Electrical and Electronics Engineers, Inc.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services in New Products, Product Summary, MicroReview, MicroNews, and MicroLaw, reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the Computer Society of the IEEE. Send editorial correspondence to *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All submissions are subject to editing for style, clarity, and space considerations. *IEEE Micro* subscribes to The Computer Press Association's code of professional ethics. BPA

From the Editor-in-Chief

Recovery

Business is better. Even the most conservative computer and semiconductor industry-watchers concede that things are much better than many December 1986 predictions. While we are not in one of the several "boom times" that many of us have experienced over the years, product shipments and new orders are on the way up. As expected, most of the companies that continued to carry out quality work and improve technology during the recession are doing very well now.

During NCC this June in Chicago, your editorial board held its annual meeting. I would like to give you a condensed account of this meeting.

- **Best *IEEE Micro* article of 1986.** "A Microprocessor-based Hypercube Supercomputer" by John P. Hayes, Trevor Mudge, Quentin F. Stout, Stephen Colley, and John Palmer, October 1986, was selected the best article of last year.
- **Magazine content.** It was agreed to update the *IEEE Micro* Author's Guide to give authors more writing direction as well as logistic guidance. Our referee review forms also need updating. Further, we will issue and publish as soon as possible revised guidelines for guest editors, based upon our new editorial calendar. At the request of some readers during a telephone survey, Associate Editor-in-Chief Joe Hootman has begun to investigate the feasibility of initiating a new "Benchmarking MPUs" column. If the concept appears viable, we would publish the result once or twice a year. One reader also suggested that we expand this effort to cover DSPs and graphic devices on a periodic basis.
- **Editorial calendar.** It was suggested by Managing Editor Marie English that we attempt to set up a calendar that contained two years of proposed themes, instead of one as has been our practice. The result is published on the inside back cover of this issue. In addition to the guest-edited special issues, the Editorial Board will solicit individual articles for "minispecial issues" containing two or three articles. The topics selected are MPU-biological interfaces, micro-transducers, personal instruments, memory technology (semiconductor, optical storage, mass storage), CAD, LAN, imaging, and CIM (two articles).

  This list is not all inclusive. If you have an article on an appropriate *IEEE Micro* topic that we haven't listed, we will be glad to review it.
- **Various.** We also discussed several other topics such as new Editorial Board nominations, reappointments, circulation, and advertising.

As usual, your comments and suggestions are encouraged. Compliments let us know when we are doing a good job for you. Constructive criticism helps us to do even better.

Those interested in more information on the TRON project can write to Dr. Ken Sakamura, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.

Have a safe summer (or winter to those readers in the southern hemisphere).

Regards,

Jim Farrell

Mailbag

We received 27 response cards in the mail for this issue. While all circled the Reader Interest Survey numbers, only a few chose to write comments this time. (Note: The Reader Interest Survey numbers are compiled periodically, and we use the tally to evaluate our columns and individual articles.)

"I liked...the MC68824 LAN controller and FDDI articles." R.S.S., Rajasthan, India

"I liked all TRON articles...would like to see more on software standards... please follow up on TRON...tell (me) how to get copies of standards...." J.W., Harvest, AL

"What is an ASIC?" J.A., Fairfax, VA (An ASIC is an application-specific integrated circuit.—JF)

"More DSP hardware." D.R., Buenos Aires, Argentina

"I liked all articles on DSP chips...." R.A., Como, Italy

"Why can't US manufacturers work together as the Japanese are doing on the TRON project?" V.S., Oak Lawn, IL

"Product Summary is excellent...." P.F., Schenectady, NY

"(TRON) Another good issue...." B.H., Albuquerque, NM

"TRON project very interesting. It is almost a 'pure' high-interest issue. Bravo!" A.B., Mexico City, Mexico

"More should be done to make (people) aware of Japanese industrial potential. You do a good job." P.S., Redondo Beach, CA

"More PC applications for electromedical systems." P.S., Martinez, Argentina

THE NEW GENERATION OF MICROPROCESSORS

Kenneth Majithia
IBM Development Laboratory

Since their inception in the early 1970's, microprocessors have changed drastically in every imaginable way. The low-performance and simplistic architectures of the seventies have evolved into the complex, powerful engines of today. These engines exhibit almost all of the essential mainframe attributes—except the price.

The need for higher and higher performance drives today's microprocessor architectures to mimic that of the mainframe. The popular approach among the manufacturers has been to integrate critical peripheral functions on chip to better emulate system-type performance. Initial 32-bit architectures were restricted by the available technology and package pin counts. However with the advent of the submicron geometries of the advanced CMOS processes, today's 32-bit microprocessors pack an unprecedented number of functions onto a single chip. Manufacturers unquestionably and unanimously embrace the on-chip memory management unit, or MMU. Other functions such as instruction caches, data caches, buffers, and floating-point blocks are becoming popular choices of the designers for on-chip integration.

The catalyst for this latest high-integration, high-performance spree has been the systems houses (DEC, AT&T, and Hewlett-Packard), each of whom instigated mainframe architectures for their own proprietary microprocessors. The general-purpose MPU vendors such as Intel, Fairchild, National, Motorola, and AMD have responded to the pressures and competitions from these systems houses by introducing new microprocessor chips that incorporate the advanced mainframe features.

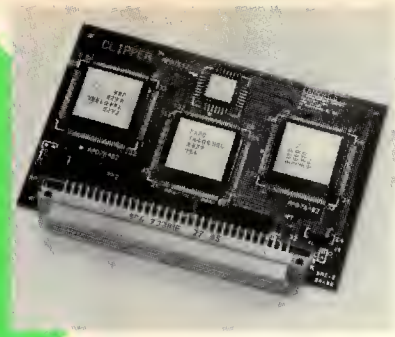
The Japanese have been equally aggressive in their new designs of high-performance microprocessors. NEC's V60 and V70 microprocessors use architectures that include not only the MMU but also an arithmetic floating-point unit on chip. Hitachi and Fujitsu have collaborated to produce a family of microprocessors adapted to the TRON operating system. These processors incorporate instruction pipelines as well as instruction and stack caches. However, unlike NEC, their FPU function is off chip.

The European manufacturer, Inmos, has begun to ship an upgraded version of its RISC-based Transputer, which contains an on-chip FPU, 4K of static RAM, and four standard communications links.

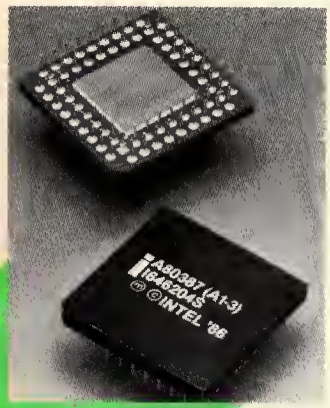
Motorola's newly announced 68030 contains an on-chip MMU. In addition it also integrates 256-byte instruction and data caches. National Semiconductor's current chip set, including the 32332, does not integrate an MMU and caches on chip. However, the new 32532 to be announced later this year contains on-board MMU and instruction and data caches.

In this special issue, three articles represent the high-performance, high-integration trend for the new generation of 32-bit microprocessors. Colin Hunter's article describes the basic architecture and configuration of the Fairchild Clipper microprocessor. The Clipper module is a three-chip set comprising a CPU/FPU, instruction cache MMU, and data cache MMU. The Clipper architecture, characterized by dual 32-bit buses to cache memory (one for instructions and one for data), essentially doubles the memory bandwidth over the single-bus architectures. Further bandwidth improvement is achieved by burst-mode updating of the caches. Other critical features—dual pipelining, error recovery and prevention—are also incorporated. Dual pipelining involves concurrent operations of the ALU and the FPU, while the error prevention/recovery feature permits contention-free operations and accurate handling of the exceptions.

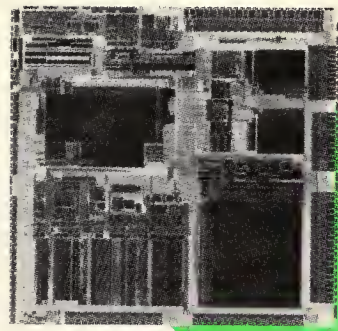
0272-1732/87/0800-0004$01.00 © 1987 IEEE



The Fairchild Clipper



The Intel 80387 coprocessor



Advanced Micro Devices Am29000

The second article by Mike Johnson addresses the systems design and performance issues involved with AMD's new 29000 microprocessor. This streamlined instruction processor extends well-publicized RISC principles into its operating system and hardware environment. A significant portion of the Am29000 architecture is implemented to enhance instruction pipeline efficiency and to provide single-cycle execution. The author also describes an application of the large, on-chip register files that is implemented on chip to improve performance. Next, he gives a detailed account of the fixed-length instruction set. Also included is a discussion of the trade-offs involved in the design of an on-chip MMU and a three-bus channel architecture. Finally, a section on exceptions and error reporting describes the manner in which errors are handled during either instruction access or data access.

The third article, authored by David Perlmutter and Alan Yuen, focuses on the newly introduced Intel 80387 coprocessor chip. This FPU provides a configuration in which the coprocessor function is not integrated on chip. The 80386 microprocessor, for which this chip is designed, was introduced in the December 1985 issue of *IEEE Micro*. The 80387 interfaces to the 80386 as a slave coprocessor through 32-bit address and data buses. The authors describe the system interface, the instruction set, and its execution, timing, and exception handling. Finally, they give an example that illustrates the application of this chip into multiprocessing and distributed processing systems.

The high-performance, high-integration approach to microprocessor architecture will continue to dominate design for some time to come. However, beyond this, application-specific microprocessor architectures for artificial intelligence, signal processing, and communications may proliferate.

Kenneth Majithia is a design engineer at the IBM Development Laboratory in San Jose, California. He has worked for Priam Corp. in the disk controller development area and for Atari, Inc., in the video system design department. His areas of interest and expertise include VLSI logic design, microprocessor architectures, local area networks, and buses. He has authored several technical papers on microprocessors and their application in signal processing.

Majithia received a BS in electrical engineering from San Jose State University and completed graduate work in computer science at Santa Clara University. He chairs both the IEEE Santa Clara Valley Section and the Technical Program and Steering Committees of the Systems Design and Networks Conference. He holds membership in Tau Beta Pi, Eta Kappa Nu, and the IEEE.

Questions concerning this introduction can be directed to Kenneth Majithia, IBM Development Laboratory, Dept. E80/862, 5600 Cottle Road, San Jose, CA 95193.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 162 Medium 163 Low 164

INTRODUCTION TO THE CLIPPER ARCHITECTURE

Colin B. Hunter
Hunter Systems Inc.

This 32-bit, three-chip module combines RISC and CISC features with mainframe-style cache and three-phase pipeline designs.

The Fairchild Clipper is a high-performance, 32-bit microprocessor whose architecture combines aspects of a reduced instruction set computer with those of a complex instruction set computer. It has features derived from supercomputers and mainframes, especially mainframe-style cache design.

Implemented on three chips, the Clipper architecture includes a CPU chip containing one integral floating-point unit and two cache-plus-memory-management units. One CAMMU processes instructions and one processes data. The Clipper architecture integrates all computational functions onto the CPU chip and all memory access functions onto the two CAMMUs. The three chips on this module replace up to 40 chips found in designs with other microprocessors. These chips are integrated onto a 3 × 4.5-inch compute module that has an effective performance approximately equal to a VAX 8600.

Figure 1 contains a diagram of the Clipper module that also shows the subunits within each chip. The CPU chip contains both a three-stage, pipelined integer execution unit and an IEEE standard floating-point unit. Each CAMMU contains an MMU (with associated 128-entry translation lookaside buffer), a 4K-byte cache, and cache control logic. Floating-point calculations can be processed in parallel with integer instructions, and address translation occurs concurrently with cache lookup.

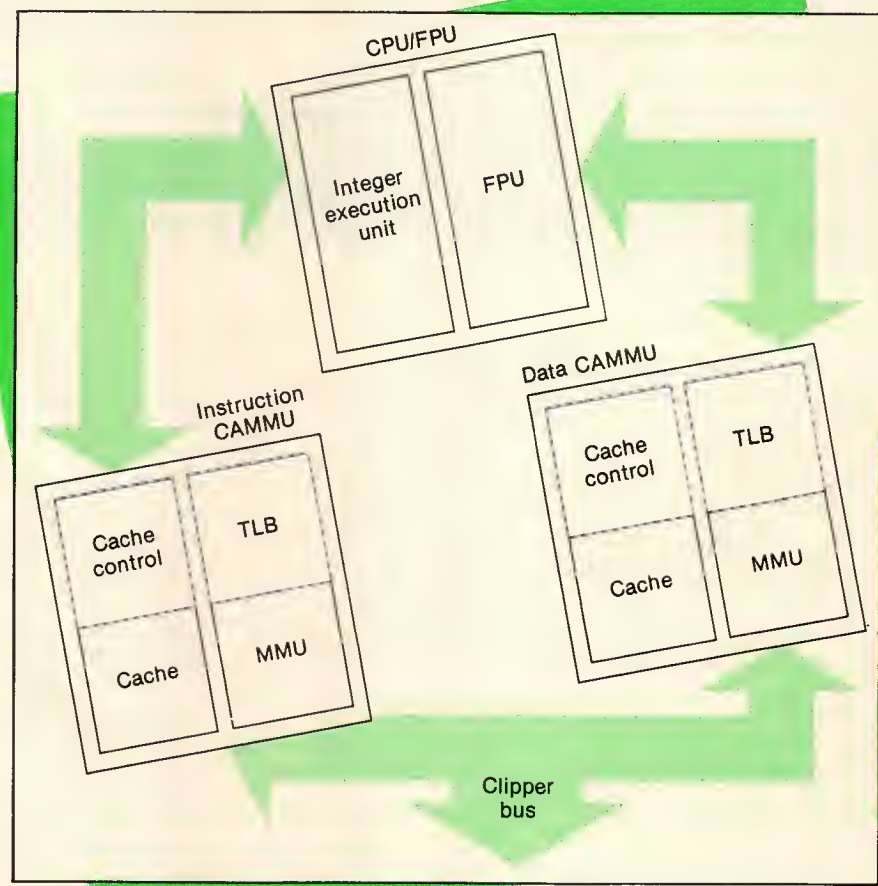
The Clipper architecture is actually three separate architectures, each defined by the boundary between the system levels and designed with different goals in mind. The innovative internal and instruction set architectures were designed for performance, that is, program execution speed. The application architecture was designed for compatibility with the installed base of software. Such compatibility was not possible until recently when Unix System V emerged as a de facto standard operating system and application programs were written in a high-level language.

As a result of this change, workstation computer manufacturers have been freed from the constraint of instruction set compatibility with older architectures. Application architecture compatibility now assumes paramount importance. The Clipper application architecture is defined by the Clipper Unix System V implementation and by the high-level languages available on the Clipper (C, Fortran, and Pascal).

Because the Clipper application architecture is standard, we won't discuss it here; we focus instead on the instruction set and internal architectures. It is important to note, however, that the Clipper hardware supporting Unix and

0272-1732/87/0800-0006$01.00 © 1987 IEEE

Figure 1. Block diagram of the Clipper module.



high-level-language compilers has had a profound effect on the design of the lower architectures.

Internal architecture

The Clipper's exceptional performance is primarily due to its advanced internal architecture. This architecture comprises a number of features, which for explanatory purposes we group into two categories:
caching with

- dual 4K-byte caches (instructions and data),
- quadword line buffers,
- multiple caching strategies, and
- a bus watch mechanism for ensuring cache consistency;

and *pipelining* with

- the three-phase pipeline itself,
- dual execution units (integer and floating-point),
- a three-stage integer execution unit, and
- a resource management mechanism for avoiding resource contention.

The overall result of the caching features is to reduce the effective memory access time by a factor of five, thus dramatically increasing the speed of instruction fetches and data reads and writes. The result of the pipelining features is to greatly improve the number of instructions that can be executed in a given period of time, by overlapping the suboperations of successive instructions.

The factor that limits the performance of most microprocessors is bus bandwidth, the amount of data per unit of time that can be transferred over the key buses in the system.[1] One of the most distinctive features of the Clipper internal architecture is the use of two buses between the CPU and the CAMMUs; one bus is used for data, the other for instructions. The use of two buses effectively doubles the bus bandwidth in this critical region.

Caching. Caching is a technique pioneered by mainframe computer designers for matching high-speed CPUs with lower speed, lower cost memory systems. The technique basically employs a small, high-speed buffer that is positioned between the CPU and the main memory. Frequently used data and instructions are loaded into this buffer where they can

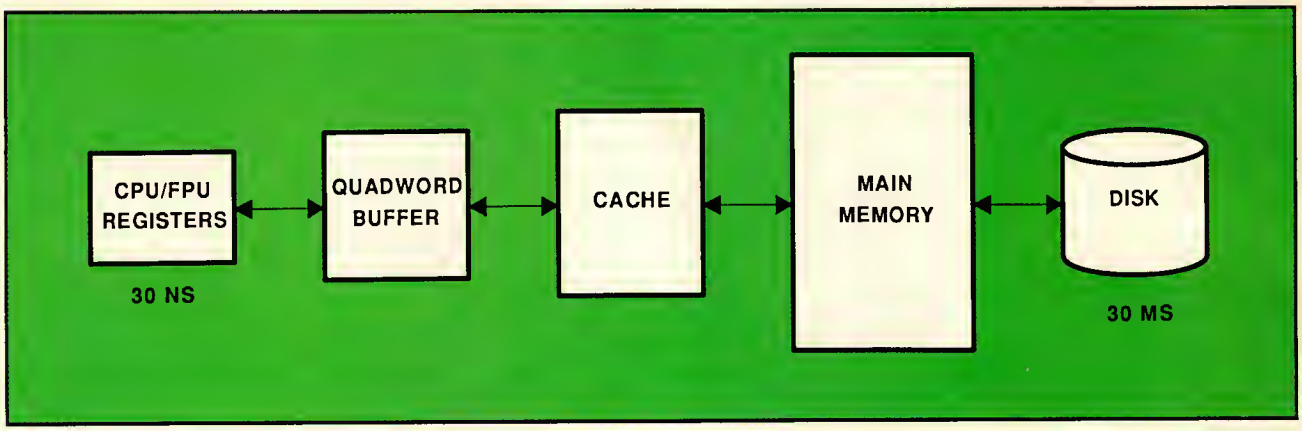


Figure 2. Memory hierarchy.

be accessed quickly; only when new items are fetched into the buffer does the main memory have to be accessed. Thus the effective memory access time is greatly reduced.

Since the Clipper clock runs at 33 MHz, the Clipper CPU is capable of initiating a memory access every 30 ns. While it is of course possible to build a memory system using high-speed static RAMs, most microprocessor systems use slower and much less expensive dynamic RAMs. The typical access time for currently available DRAMs is around 120 ns, with a conservative memory bus design bringing the total memory access time to around 500 ns. The Clipper uses a cache memory to bridge the gap between its 30-ns CPU and the 500-ns main memory. Caches are thus part of the memory hierarchy of the Clipper. (See Figure 2.)

At one end of the memory hierarchy are the Clipper's registers, which have an access time of one clock cycle, or 30 ns. At the opposite end of the hierarchy is the mass storage system—usually a hard disk—with an access time of around 30 ms. Bridging the gap of six orders of magnitude (a factor of a million) between the registers and the disk is the main memory and caches. The Clipper actually employs two levels of caches: an 8K-byte main cache and a very high speed cache-within-the-cache called a *quadword, or line buffer.*

The effectiveness of a cache system depends on three factors:

- the *cache access time,* the number of clock cycles required to access data/instructions contained in the cache;
- the *hit rate,* the frequency that the desired data/instruction is actually found in the cache; and
- the *replacement time* on a cache miss, the number of clock cycles required to fetch an item from main memory into the cache when a cache miss occurs.

The effective access time is the average time taken to access data or instructions. It depends on the interaction of the three factors mentioned above and the character of the program itself. Most programs tend to reaccess memory locations soon after they are accessed for the first time, or else to access nearby locations. This behavior is called *locality of reference.* The more locality a program exhibits, the more effective caching is at reducing effective access time. For typical programs, the Clipper caches reduce the effective access time from 500 ns to around 100 ns, a 500-percent improvement in performance.

Cache access time. The Clipper's cache memory actually consists of two 4K-byte caches: the instruction cache and the data cache, each located on a CAMMU. Each CAMMU also contains a 16-byte (quadword) line buffer. The data in the caches is organized as 256 lines, with each line containing 16 bytes. Whenever an access is made to the cache, the entire line containing the accessed item is loaded into the line buffer. Subsequent accesses to the same line will not require that the cache be accessed; instead, the request will be satisfied from the line buffer.

The line buffer can be accessed in one clock cycle (30 ns). If the line buffer misses, the cache can be accessed in an additional two clock cycles, for an access time of three clocks, or 90 ns. It takes 15 ns to transfer the address over the bus to the CAMMU. (The CPU and CAMMU are designed to use one half of a clock cycle for this transfer.) Similarly, it takes only 15 ns to transfer the data back to the CPU. Thus the total access time for the line buffer is 60 ns and for the main cache, 120 ns.

Hit rate. The hit rate is a function of four factors:

- the size of the cache in bytes, (obviously, a larger cache is better than a smaller cache);
- the number of bytes in a line (generally, a larger line size improves the effective memory access time);

• the degree of set associativity (the higher the degree of associativity, the more flexible the cache and the less likely useful data is overwritten); and
• the program's locality.

Since computer architects have no control over program locality, we will not discuss this factor further, except to observe that modern structured programming techniques are very beneficial in this regard. Other things being the same, a larger cache is better than a smaller cache; but other things are seldom the same. Caches are also characterized by their line size and set associativity.

The line size is the number of bytes that are fetched into the cache when main memory is accessed. The more bytes that are fetched, the fewer main memory accesses have to be made, if programs have reasonable locality. The Clipper uses a 16-byte line size, which means that 16 bytes are updated when a cache miss occurs, and 16 bytes are transferred whenever main memory is accessed. Each cache contains 256 lines.

If the cache is full and a cache miss occurs, the new line overwrites a line of data already in the cache. The set associativity of the cache determines how much flexibility the cache has in determining which line to overwrite. The simplest kind of cache is a *direct-mapped cache,* in which each main memory location can be written into only one cache location. A two-way set-associative cache can write each memory location into two cache locations. Similarly, four cache locations are available in four-way set associativity, and so on. Figure 3 illustrates the Clipper's two-way set associativity.

In the Clipper cache, each set contains two lines of 16 bytes, and there are 128 sets. The collection of all the left-hand lines is called the W compartment, the collection of right-hand lines is called the X compartment. Each compartment contains a total of 2K bytes. Physical memory is divided into 4K-byte pages, and within each page, a location is mapped into a unique location in both compartments. Consequently, several main memory locations are mapped into the same pair of cache locations. The first time

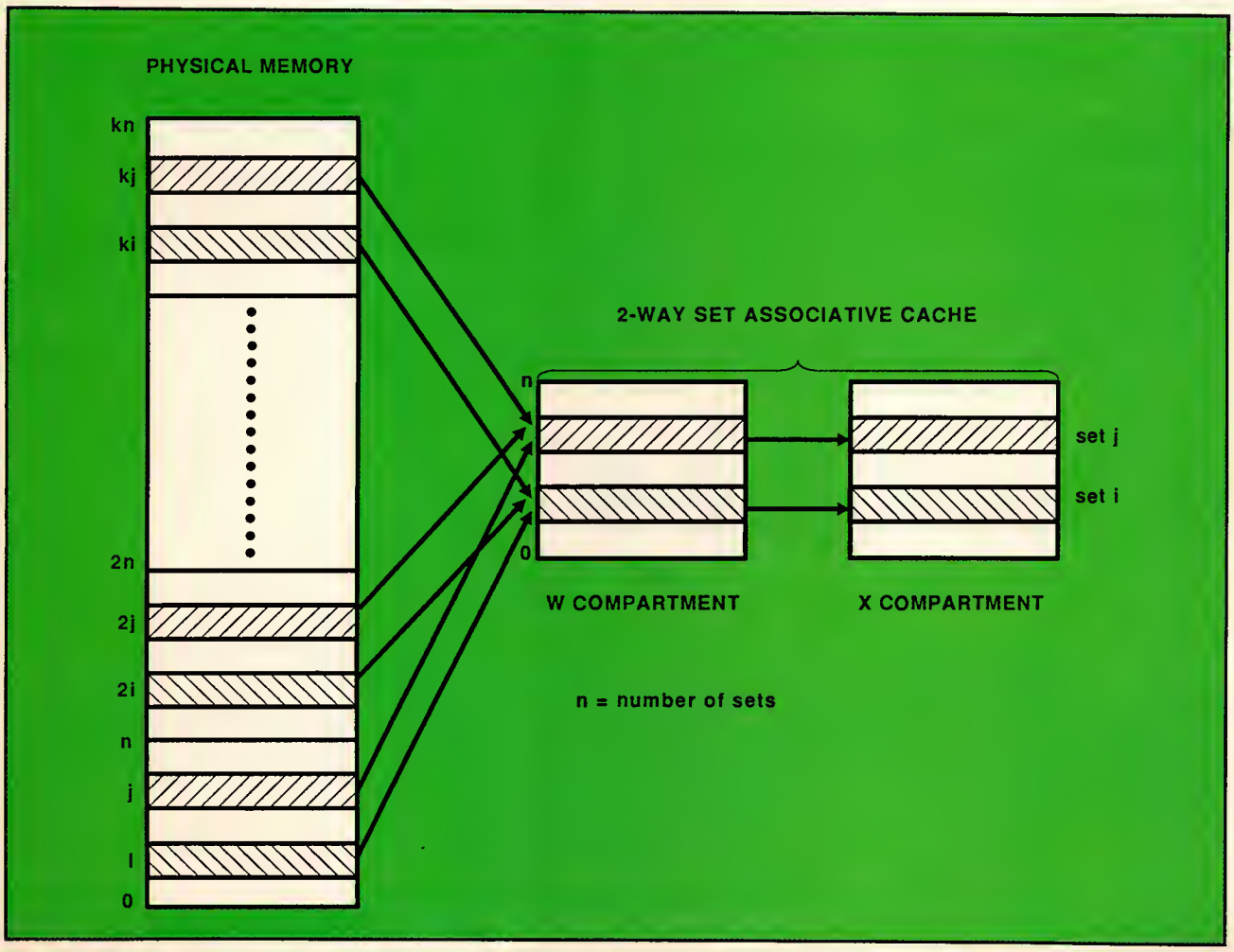


Figure 3. Set associativity.

main memory is accessed, 16 bytes are fetched and stored in a line of the W compartment. If a new main memory access requires that line to be overwritten, the cache instead employs the corresponding line in the X compartment. If the corresponding lines of both compartments are full, the cache overwrites the one that was least recently used.

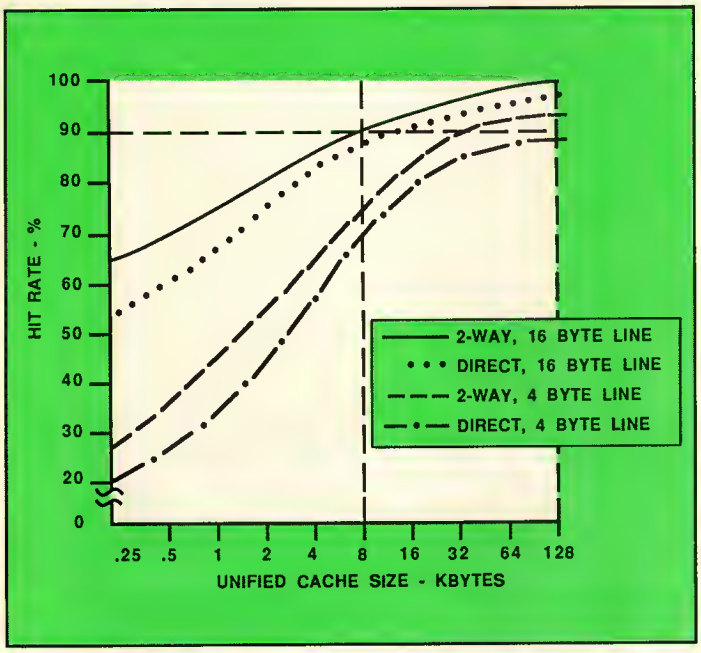


Figure 4. Cache hit rate.

Thus we can see that the efficiency of a cache depends on both the line size and set associativity, and not just the size of the cache. Figure 4 shows how the hit rate varies with cache size, line size, and associativity.[2] Notice that the Clipper's 8K-byte total cache size along with its 16-byte line size and two-way set associativity yield a hit rate that is about the same as a 128K-byte, 4-byte line, direct-mapped cache.[3,4]

In fact, the data cache has a hit rate of greater than 90 percent, while the instruction cache's hit rate exceeds 93 percent. And when prefetch is enabled, the instruction cache can exceed a 96-percent hit rate. Prefetching is a mechanism available on the instruction CAMMU for bringing into the cache the next 16 bytes of memory before they are accessed. The CAMMU maintains a copy of the instruction pointer register and uses it to fetch instructions before the CPU requests them. For in-line code sequences, the prefetch mechanism ensures a 100-percent hit rate, since prefetching is performed concurrently with other CPU and CAMMU operations.

Replacement time. Inevitably, a cache miss will eventually occur. The miss is the number of clock cycles that occur from the time the miss is detected by the CAMMU until the last byte of the line is loaded into the cache. Leaving aside the speed of memory, this time is determined by the bus clock rate, the line length, and the number of clock cycles required to transfer a line over the memory bus.

The Clipper's memory bus runs at a clock speed of 16.6 MHz (that is, one bus cycle is equal to two CPU

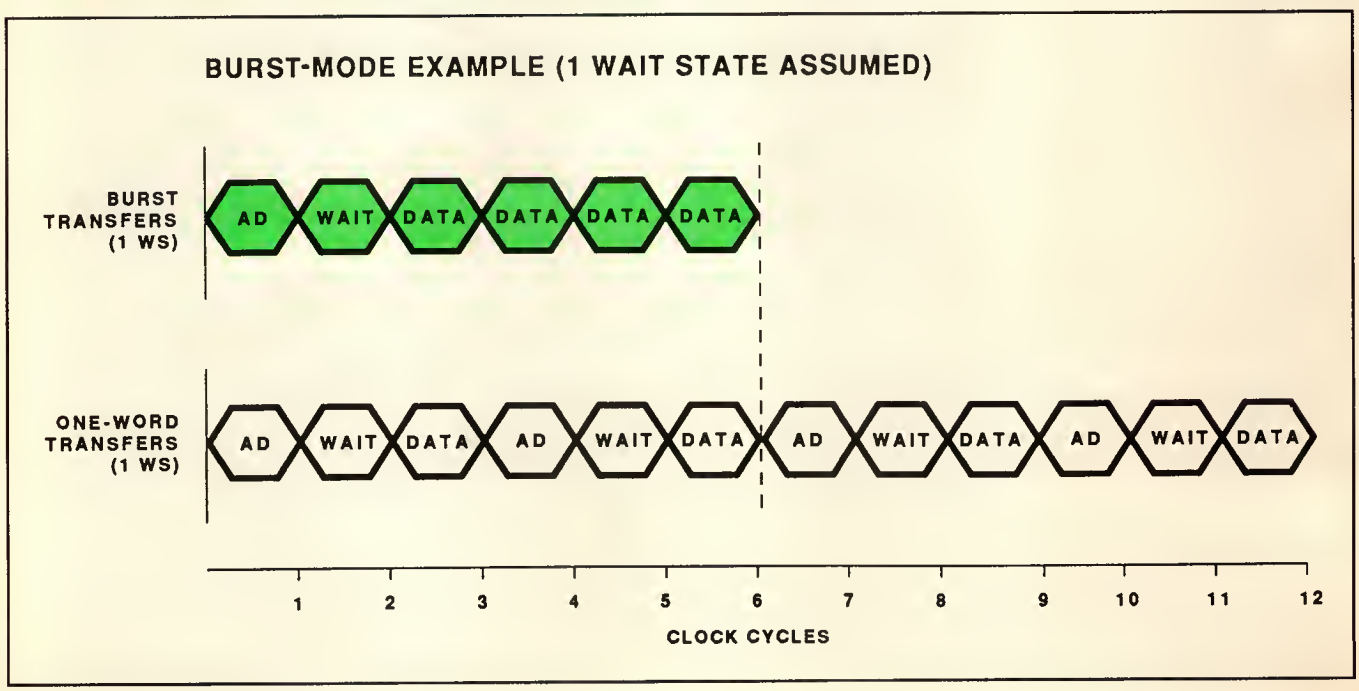


Figure 5. Cache replacement time.

cycles). Since the line length is 16 bytes, the replacement time depends on how many clock cycles it takes to transfer 16 bytes over the bus.

The Clipper employs a special supercomputer-derived technique called *burst mode* to transfer four words (16 bytes) in a single memory access. Since only one access is made, only one address is transferred to memory (the starting address of the line of 16 bytes), and only one wait state is needed. The next four clock cycles transfer the four words of the line. Burst-mode transfers of the entire line are twice as fast as conventional four-word transfers. The top part of Figure 5 contrasts burst-mode timing with the timing achieved by conventional 32-bit microprocessors.

The concepts of hit rate and miss replacement are relatively straightforward for read accesses: When a cache hit occurs, the data is used; when a miss occurs, an access is made to main memory. Write accesses are more complicated, because a strategy must be developed to update main memory. The Clipper supports two main cache write strategies: *write through* and *copy back*.

The user can designate any page in memory as cacheable or noncacheable, and if cacheable, select one of the two caching strategies for that page. In addition, pages can be given protection attributes, such as Read Only.

Under the simplest write strategy, called Write Through, main memory is updated every time the cache is altered. Since the cache and main memory are changed at the same time, they always contain the same data, and main memory is always up to date. The penalty is that little or no benefit is derived from the cache for writes; thus the cache is only about half as effective as it might be. The designers of the Clipper felt that an exclusive reliance on Write Through seriously limits performance.

Under the copy-back strategy, memory is updated only when a line that has been modified in the cache needs to be overwritten by another line. At that point the old, modified line is "copied back" to memory before it is overwritten. The copy-back strategy minimizes memory accesses and thus provides the highest performance, but this strategy requires more hardware support. The designers of the Clipper built the needed hardware support mechanisms into the CAMMU, so the user can use copy-back mode whenever high performance is required.

The *bus watch* mechanism in the CAMMU ensures data consistency between cache and main memory, so the copy-back strategy can be employed successfully. The basic problem that bus watch solves is that of getting old, out-of-date data (also called stale data), a problem that can occur when multiple bus masters access memory independently.

Bus watch eliminates this problem by automatically ensuring consistency. The bus watch mechanism fulfills read requests by other bus masters from the cache instead of memory, if there is any difference between the two. It also ensures that writes by other bus masters update the cache as well as main memory. Bus watch is controlled by setting bits in the CAMMU control register. Figure 6 shows the way bus watch works.

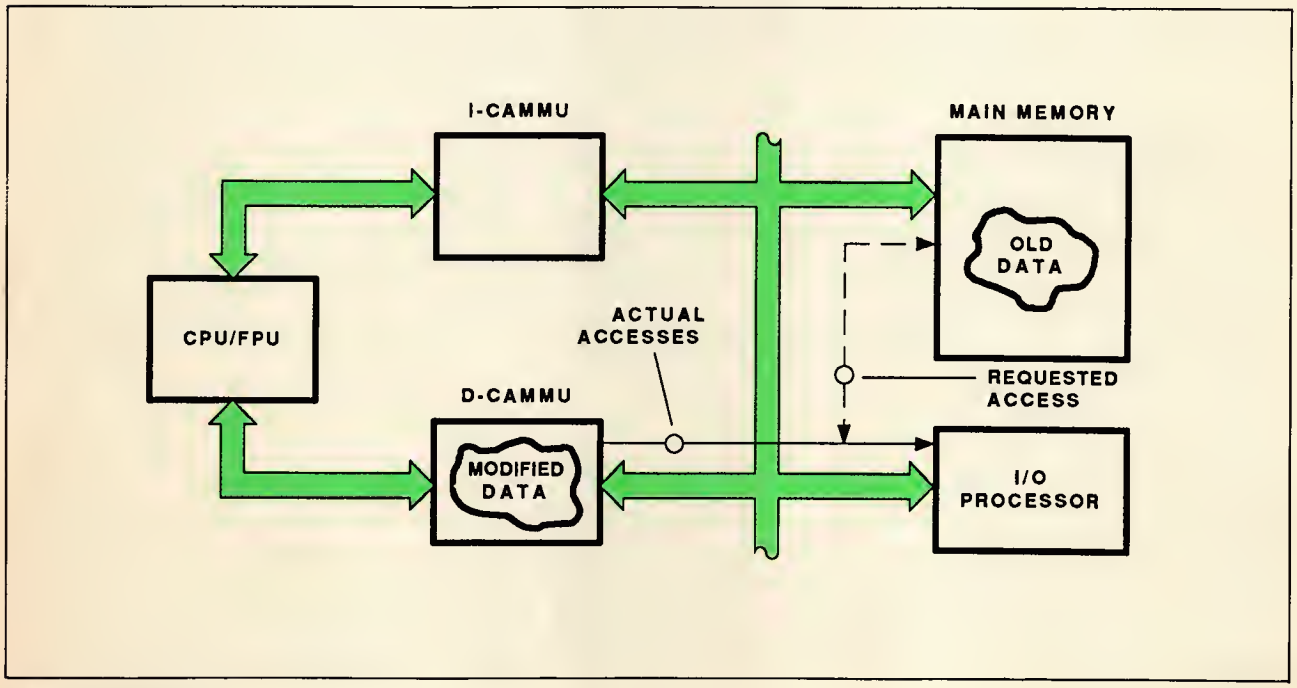


Figure 6. Bus watch.

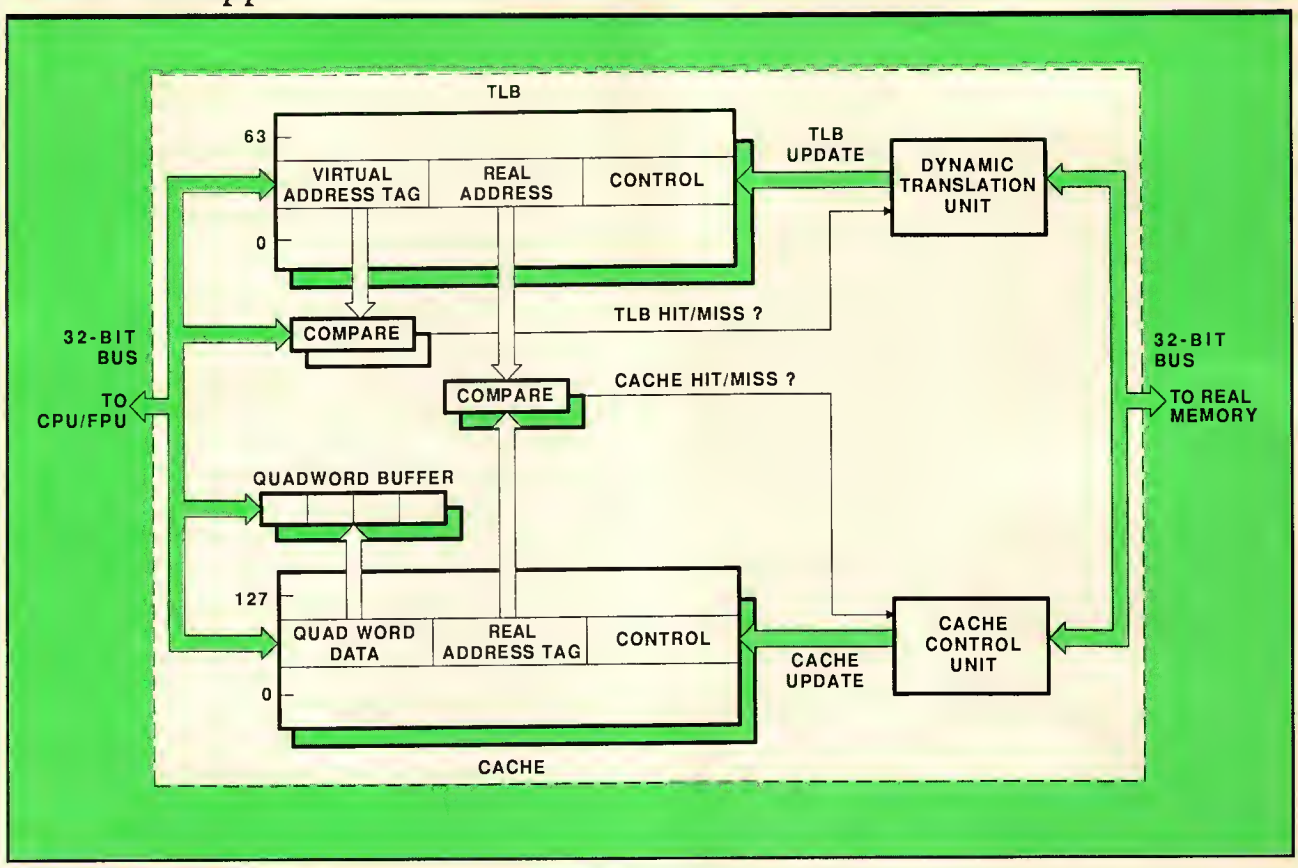


Figure 7. Cache access mechanism.

Table 1.
CPU process for generating a virtual address.

Step	Description
1	If it is an instruction access, the virtual address is sent to the instruction CAMMU; a data address is sent to the data CAMMU.
2	The virtual address is compared with the value stored in a buffer called the *virtual address cache*. This buffer contains the last virtual address sent to the cache. (The line buffer contains the last line accessed, one of whose bytes was pointed to by the last virtual address.)
3	If the two values match, we have a line buffer hit, and bits 2-3 of the virtual address are used to select one of the four data words in the line buffer to be returned to the CPU. (Actually, each CAMMU contains two line buffers, one for each compartment, but the most recently accessed one is used.)
4	If the two values do not match, the line buffer misses, and the CAMMU proceeds concurrently to translate the virtual address and access the cache. The virtual address is stored in the virtual address cache.
5	The MMU/TLB translates bits 12-31 of the virtual address into a 20-bit real value. Bit 11 of the virtual address is appended, yielding a 21-bit value called the real address tag. Meanwhile, bits 4-10 of the virtual address have selected one of the 128 sets in the cache.
6	The 21-bit real address tag from the TLB is compared with the real address tag field of both lines in the selected set of the cache.
7	If the two values match for one of the two lines, we have a cache hit, and again bits 2-3 of the virtual address are used to select one of the four data words to be returned to the CPU. The line itself is loaded into the line buffer.
8	If a cache miss occurs, the cache control mechanism causes a main memory access to be generated, by appending virtual address bits 0-10 to the 21-bit real address tag to produce a 32-bit real address.
9	The main memory access causes a new line to be loaded into the cache and the line buffer, and Steps 6 and 7 are performed again.

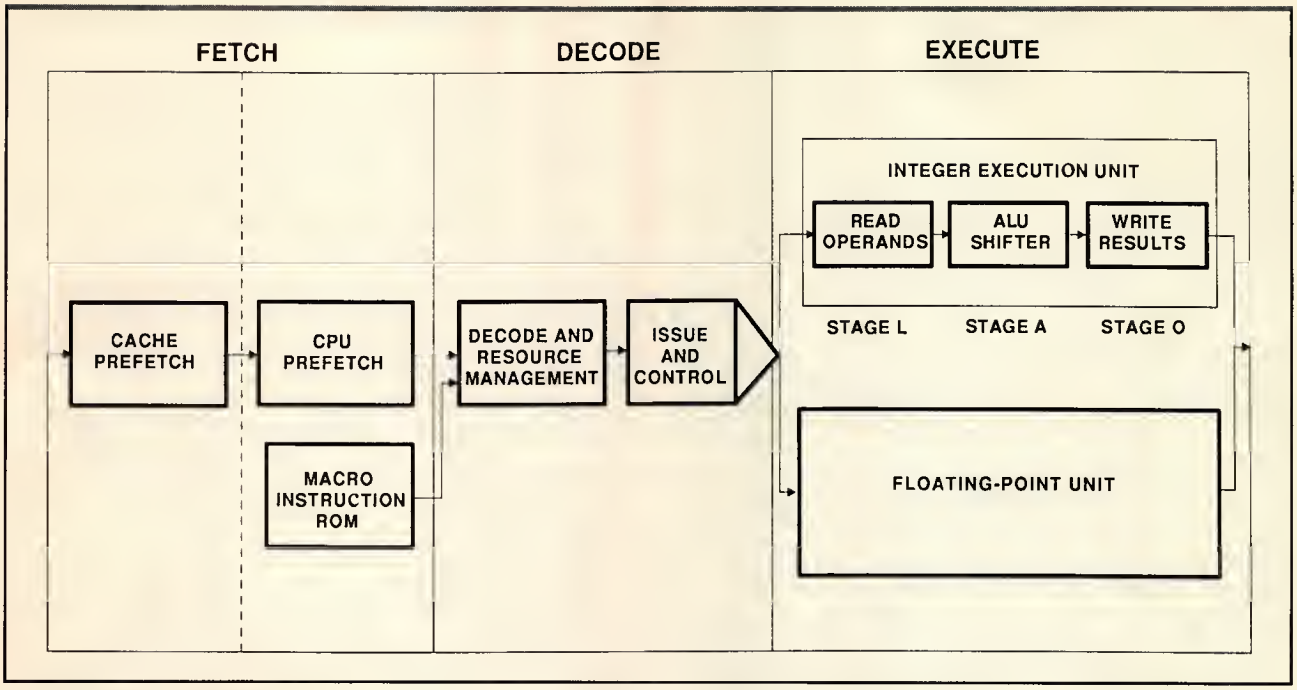


Figure 8. Clipper pipeline.

Cache implementation. The Clipper cache is closely tied to the MMU and its TLB, which we describe later. The CPU generates 32-bit virtual addresses, which are translated by the MMU/TLB into real addresses. The cache mechanism then compares these real addresses with addresses stored internally, and if a match is found, the word associated with the internal address is returned to the CPU. Figure 7 is a schematic representation of this process, assuming a cache hit on a line in the W compartment. Table 1 details the process.

Pipelining. Pipelining is a technique extensively used in supercomputers to improve performance by overlapping operations that can be performed concurrently. Most 32-bit microprocessors make use of a simple form of pipelining. By contrast, the Clipper pipeline system combines the benefits of a simple three-phase pipeline with extensive use of concurrency and parallelism *within* each phase of the pipe, especially the execute phase.

Three-phase pipeline. Figure 8 shows the three phases of the Clipper pipeline: fetch, decode, and execute. Each of these phases proceeds independently without having to wait for the first instruction to finish all phases. Since the Clipper arithmetic and logical instructions operate only on registers, there is no need for a special "address pipeline" or separate effective address-calculating phases. Those instructions that require an effective address calculation simply make one pass through the integer execution unit's ALU. Only one pass is needed because these instructions will not require the ALU for arithmetic or logical operations.

Notice that the figure shows a cache prefetch subphase to the fetch phase. This item refers to the instruction CAMMU's prefetch mechanism mentioned earlier. This prefetching helps the CPU to keep the instruction buffer full and improves the cache hit rate for the I-CAMMU.

The resource manager keeps a table of all resources and which instruction is using each one; it also maintains the status of all instructions that are currently executing. Because of this detailed tracking, the Clipper can restart instructions that cause page faults or continue instructions after interrupts and traps. Programs, interrupts, and traps cannot crash the pipeline.

Dual execution units. The CPU contains two execution units that comprise the final phase of the pipeline. The integer execution unit handles all integer arithmetic operations plus all address calculations; the floating-point execution unit processes floating-point arithmetic.

The integer execution unit is itself subdivided into three stages, L, A, and O. Each stage can proceed concurrently and can be overlapped for successive instructions. In the first stage, operands are read from the general register file into the L registers. Immediate operands are taken directly from the instruction buffer to the L registers via the J register. In the second, or A, stage arithmetic, logical, or shift operations are performed on the operands in the L registers or on intermediate results from the last operation, and the output is stored in the A register. In the final stage the contents of the A register are sent to the FPU, stored in the general register file, fed back via the bypass loop to the A stage (intermediate results), or sent out to the data CAMMU.

The bypass loop makes it possible for the intermediate results of multi-instruction calculations to be fed back immediately to the next instruction. Without this loop, the whole pipeline would have to be flushed.

Table 2.
Clipper RISC and CISC features.

Approaches	Features
RISC	Arithmetic and logical instructions operate on registers; basically only loads, stores, and branches, along with stack manipulation instructions, access memory.
	There are plenty of registers: thirty-two 32-bit registers, 16 for the operating system and 16 for the user.
	The instruction format is designed so that all instructions are simple multiples of one basic size (16 bits), and the most frequently used instructions are shortest.
CISC	Instructions for access memory are provided with a complete set of addressing modes that facilitate access to high-level data structures.
	Explicit support for stacks and strings is provided.
	Separate modes of operation are provided for the user and the operating system, each mode having its own resources and privileges.
	Explicit support is provided for key operating system functions, such as system calls, exception handling, and virtual memory.

The instructions from the macro instruction unit make use of their own register files, so they do not require general register resources being used by ordinary instructions. In addition, they can use all the other registers on the CPU.

The FPU performs single- and double-precision floating-point operations concurrently with the integer execution unit, using its own ALU and set of eight 64-bit registers. Because the FPU is on the CPU chip, floating-point operations do not require additional bus transfers; thus bus traffic is reduced, and performance is improved. All Clipper floating-point arithmetic operations conform to the IEEE 754 standard.

Programming model

A primary design objective of the Clipper was to provide an instruction set architecture that was well suited to operating systems and compilers. In this context, well suited means that the architecture facilitates two somewhat divergent goals:

- The instructions should execute rapidly, preferably in one per clock cycle; and
- It should be easy to write programs using the instructions, especially operating systems, and easy for compiler code generators to target them.

The first goal pushes the designer toward a very simple instruction set, with few higher level concepts supported in the hardware. This approach is often called the reduced instruction set computer, or RISC, philosophy.[5] The second goal pushes the designer in quite a different direction—toward a richer instruction set with direct support for concepts from high-level languages and operating systems. This latter approach has several names—language-directed architectures, high-level architectures, and so on—depending on the concepts supported, but we will refer to them all as the complex instruction set, or CISC, computer philosophy.[6]

The Clipper instruction set is a balance between the RISC and CISC approaches, a balance which is also a characteristic of supercomputer architectures. In more detail, the Clipper instruction set architecture derives the features described in Table 2 from the two approaches.

Registers. In addition to the program counter (PC) register, most architectures provide several other registers that are used as fast local storage for addresses and data. The width of these registers is determined by the size of the typical unit of data manipulated by the processor, the *word size*. Since addresses are stored as well as data, and since the biggest address is determined by the PC size, it is usually convenient for the word size to be the same as the PC size and for storage registers to have the same size as the PC.

The Clipper architecture has one 32-bit PC and thirty-two 32-bit storage registers, more than any other commercial microprocessor. In addition, the Clipper contains eight 64-bit storage registers that are dedicated to floating-point arithmetic. The 32-bit registers are completely general purpose; each one can be used to store either an address or a word of data. The use of general-purpose registers, instead of special-purpose data and address registers, is preferred because it eliminates unnecessary register-to-register transfers when address arithmetic must be performed.

The Clipper has two operating modes, user and supervisor, distinguished by the instructions they are permitted to execute and the registers they can use. A program executing in supervisor mode (usually the operating system) can access data in all 32 general-purpose registers and all eight floating-point registers. User-mode programs can only access 16 of the general-purpose registers (called the *user registers*) and the floating-point registers; the 16 registers that are inaccessible to user programs are called the *supervisor registers*.

Figure 9 is a diagram of all the registers in the Clipper module, including both the CPU/FPU registers and the CAMMU registers.

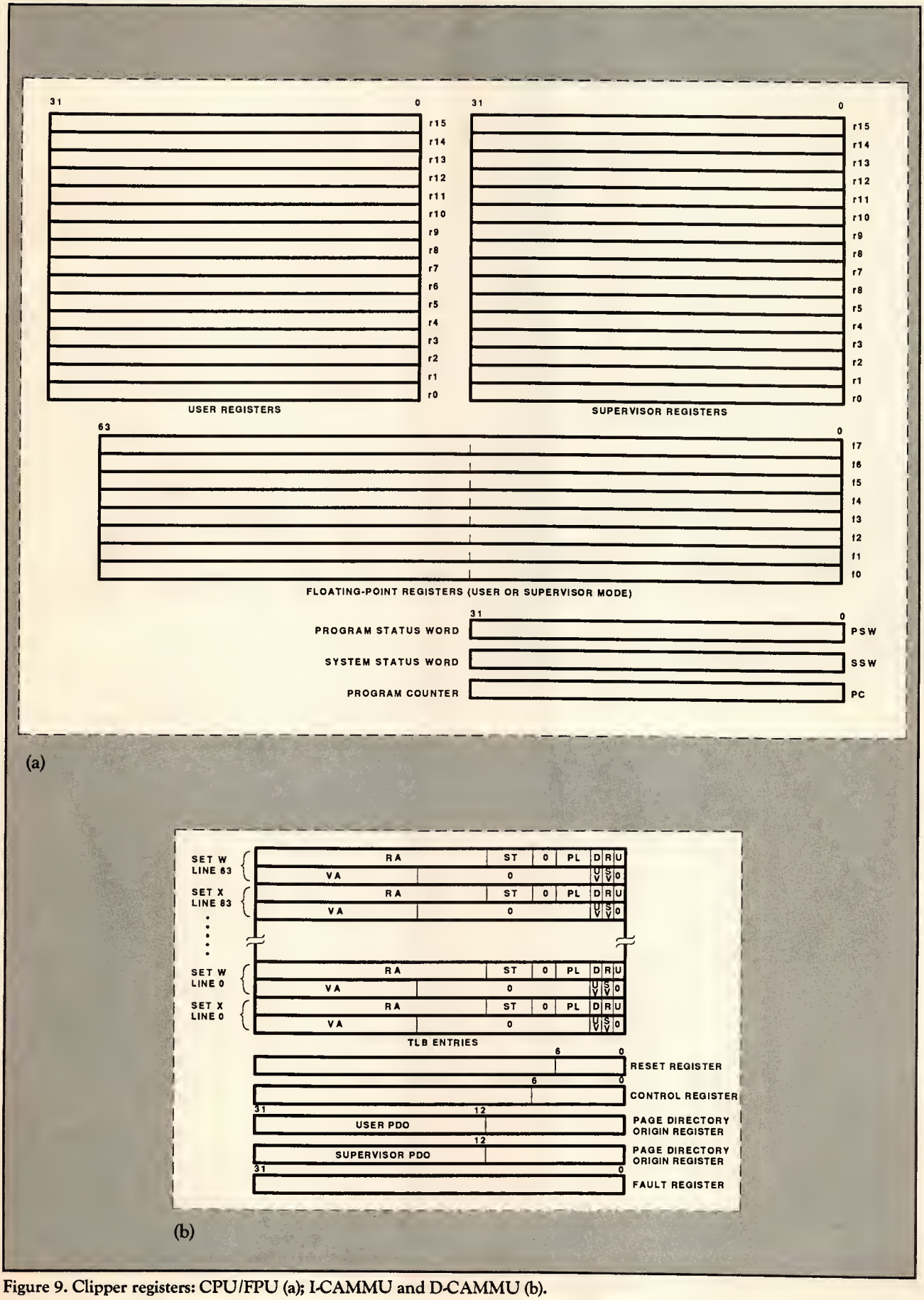


Figure 9. Clipper registers: CPU/FPU (a); I-CAMMU and D-CAMMU (b).

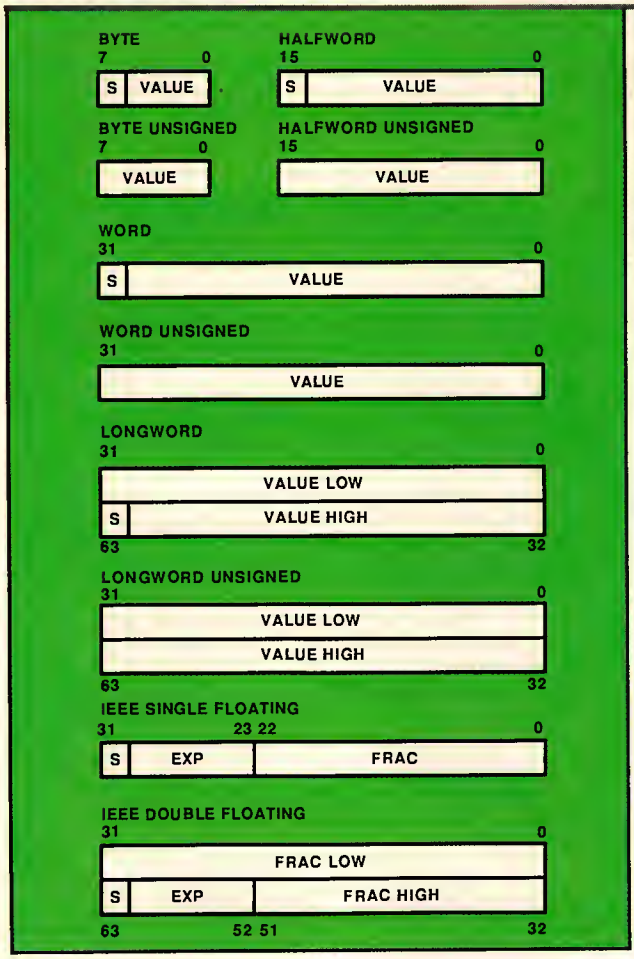


Figure 10. Primitive data types.

Besides the PC, general-purpose registers, and floating-point registers, the Clipper CPU/FPU has two 32-bit status words, the PSW and the SSW. The status words contain bits, called *flags*, which identify and control the CPU's state. The PSW, which is accessible in either mode, primarily contains flags that identify exception conditions (described later). The SSW, which is accessible only in supervisor mode, contains bits that control interrupts, address translation and protection, and mode of operation.

Each CAMMU contains five software-accessible registers, which are used for initialization and control. Two of these registers (supervisor PDO and user PDO) are used in address translation; they contain the base addresses of the supervisor and user page table directories (detailed later). The fault register is loaded with the virtual address associated with certain fault conditions; it can be used by the operating system to support virtual memory. The control register and reset register control the CAMMU.

Data types. Many applications that require a high-performance computer are calculation-intensive, number-crunching problems. In these applications 32-bit integers are often used because variables of this length provide more precision than 16-bit integers. (A 32-bit integer contains the equivalent of 10 decimal digits.) A key requirement for high-performance microprocessors is therefore the capability of manipulating 32-bit integers; that is, they need operators such as Add and Multiply that directly (in one instruction) handle 32 bits.

The Clipper supports 10 data types (shown in Figure 10). Signed and unsigned bytes, half words (16 bits), words (32 bits), and long words (64 bits) are available, along with single-precision (32-bit) and double-precision (64-bit) IEEE Standard floating-point numbers. These primitive data types can be used to build complex structured data types, such as arrays and records, that are typically found in high-level languages. Clipper provides several addressing modes that facilitate accessing structured data types.

Instructions. The Clipper instruction set contains 101 hardwired instructions and 67 macro instructions for operating on the basic data types. Each instruction specifies an operation to be performed, and the type and location of the operands. These operands can be located either in memory, in a register, or in the instruction itself. Rapid instruction decoding is achieved easily, since all instructions are built of half-word units called *parcels*. Depending on the instruction, from one to four parcels can be used.

The instruction formats fall into two groups: those with addresses and those without. The former are the instructions that need to access memory (primarily loads, stores, and branches); the latter are the arithmetic/logical instructions that generally execute in a single clock cycle. Clipper instructions have zero, one, or two operands, but only one operand can be accessed by a memory address.

The Clipper instruction set consists of the 10 categories of instructions listed in Table 3.

The 67 macro instructions are implemented in the macro instruction unit as sequences of hardwired instructions. Except for their distinctive format, nothing distinguishes the macro instructions from hardwired instructions as far as the programmer is concerned. The macro instructions are scattered over the 10 categories; for example, all conversions and string instructions are macros, along with most stack instructions (except Push and Pop), and also one or two load/store, move, arithmetic, and control instructions. Six instructions (all macros) are *privileged*; that is, they can only be accessed by a program in supervisor mode.

Addressing modes. If an operand of an instruction is in memory, several alternative ways, called addressing modes, exist for accessing it. An addressing mode is basically just a way of specifying a virtual address as the sum of several factors, which can be stored in registers or provided with the instruction itself. The Clipper supports nine memory addressing modes, which are shown diagrammatically in Figure 11.

Table 3.
Clipper instruction categories.

Category	Function	Category	Function
Load/store	Transfers addresses, bytes, half words, words, long words, and floating-point quantities between memory and registers	Conversion	Converts floating-point numbers of both precisions to integers using the IEEE rounding modes
Move	Transfers 32- and 64-bit quantities between registers	Compare	Compares the values of words and both precisions of floating-point numbers; atomic test-and-set instructions also available
Arithmetic	Add, Subtract, Multiply, Divide, Negate, Modulus, and Scale (multiply by a power of 2) supported for registers or variable-length immediate values	String	Manipulates character strings; Compare, Initialize, and Move provided
Logical	And, Or, Xor, and Nor for registers	Stack	Manipulates the stack; Push and Pop, along with save multiple registers and restore multiple registers available
Shift/rotate	Shifts arithmetic and logic (the difference lies in their treatment of right shifts for signed quantities) of both words and long words	Control	Branches, call, call supervisor, returns, and NOP provided

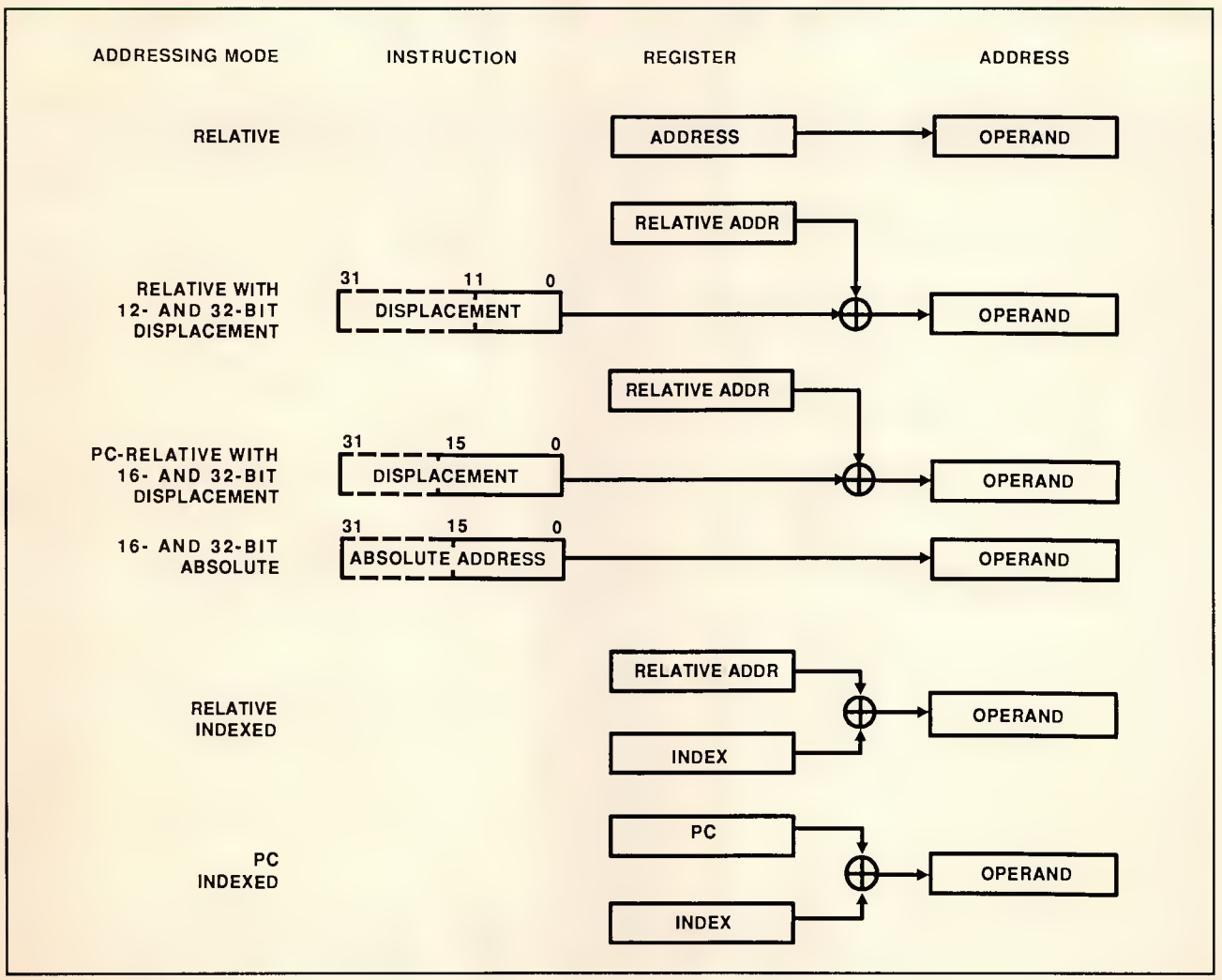


Figure 11. Addressing modes.

With the relative mode and the two relative-with-displacement modes, the virtual address is either in a register or is computed from the sum of the contents of a register and a displacement value carried with the instruction. The two absolute modes carry the virtual address as a pure displacement value with the instruction. The two PC relative modes are useful for branches relative to the current value of the PC. The two indexed modes compute the virtual address by summing the contents of two registers.

Use of these addressing modes facilitates access to the arrays and records commonly found in high-level languages.

Exceptions. Exceptions are internal hardware conditions, external events, or even particular instructions that cause the normal operation of the processor to be suspended and a special sequence of operations performed. The three basic types of exceptions are:

- *Traps*, the anomalous internal events occurring during the processing of an instruction. Classic examples include an attempt to divide by zero or a page fault in a virtual memory system.
- *Interrupts*, an external device's method of signaling the CPU that it needs servicing. For example, a DMA controller signaling that it has transferred a block of data into memory.
- *Supervisor calls,* the program-generated requests for operating system services.

If any one of these exceptions occurs, it is usually necessary to immediately invoke a software handler to respond to the exceptional condition. This need for immediate action forces exceptions to suspend normal processing—they simply can't wait. When the handler finishes its work, control returns to the point where it was interrupted.

Pipelining complicates exception handling, since the pipeline must be cleared out as soon as the exception occurs. This is necessary so the exception handler can be executed immediately after the instruction that was in the execution phase when the exception occurred. Then when the normal processing resumes, the instruction pointer must be backed up to

Real Address (Hex)	Description
Data Memory Trap Group:	
108	Corrected Memory Error
110	Uncorrectable Memory Error
128	Page Fault
130	Read Protect Fault
138	Write Protect Fault
Floating-Point Trap Group:	
180	Floating Inexact
188	Floating Underflow
190	Floating Divide by Zero
1A0	Floating Overflow
1C0	Floating Invalid Operation
Integer Arithmetic Trap Group:	
208	Integer Divide by Zero
Instruction Memory Trap Group:	
288	Corrected Memory Error
290	Uncorrectable Memory Error
2A8	Page Fault
2B0	Execute Protect Fault
Illegal Operation Trap Group:	
300	Illegal Operation
308	Privileged Instruction

Real Address (Hex)	Description
Diagnostic Trap Group:	
380	Trace Trap
Supervisor Calls:	
400	Call Supervisor 0
408	Call Supervisor 1
.	
.	
.	
7F8	Call Supervisor 127
Prioritized Interrupts:	
800	Non-Maskable Interrupt
808	Interrupt Group 0 Number 1
810	Interrupt Group 0 Number 2
.	
.	
.	
878	Interrupt Group 0 Number 15
880	Interrupt Group 1 Number 0
888	Interrupt Group 1 Number 1
.	
.	
.	
FF8	Interrupt Group 15 Number 15

Figure 12. Exception vector table.

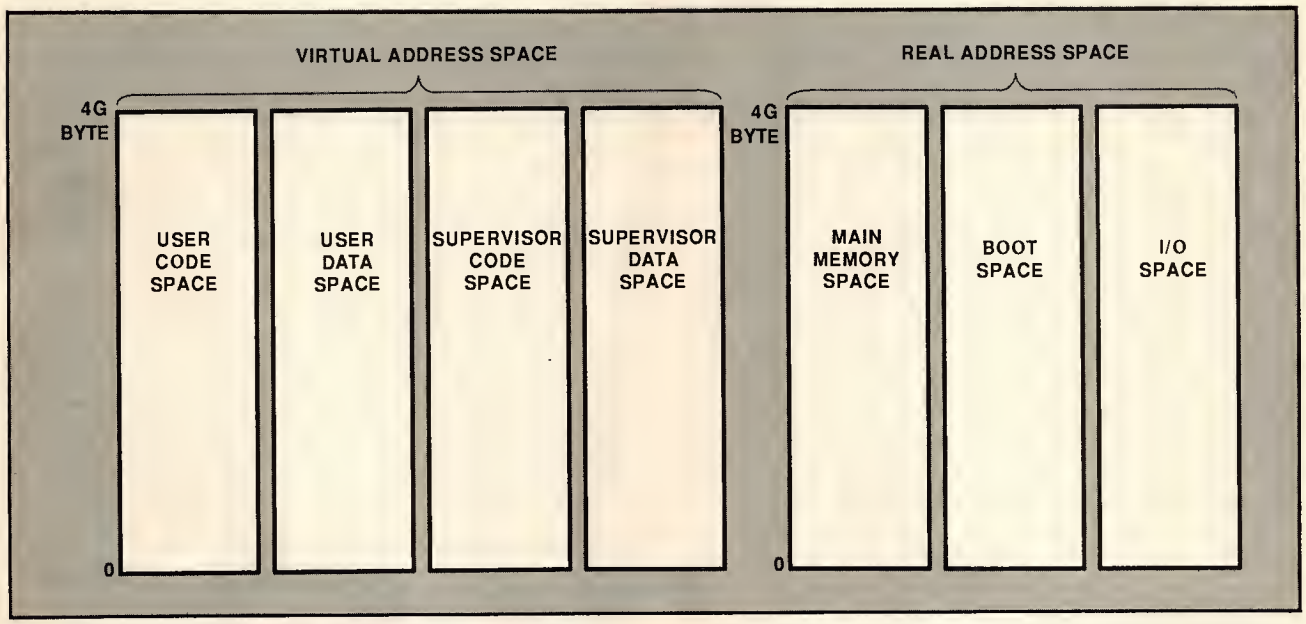


Figure 13. Clipper memory architecture.

refetch the instruction immediately following the one that was executing when the exception occurred. Multiple exceptions present additional complications; for example, a division by zero occurring during the same clock cycle as a page fault.

The Clipper architecture supports 18 traps, 256 vectored interrupts, and 128 programmable supervisor calls. The traps are caused by page faults, memory protection violations, floating-point errors such as overflow, integer arithmetic errors such as division by zero, and privileged-instruction violation by a user-mode program. When one or more of these conditions occurs, the hardware automatically generates the appropriate trap. Interrupts are signaled by activity on the interrupt pins, with the type of interrupt encoded as an eight-bit quantity on the interrupt bus. Supervisor calls are made by executing the *Calls* instruction with a parameter specifying which call.

No matter what their cause or type, all exceptions are handled in much the same way. First, the current PC, SSW, and PSW are saved on the supervisor stack, then a new SSW and a new PC are copied from a data structure called the exception vector table. The EVT, located in the first real page of memory, contains the address and SSW value for every exception handler routine. These address/SSW pairs are stored in predefined locations, with each location corresponding to a particular type of trap, interrupt, or call. Figure 12 shows the layout of the EVT.

After the exception has been handled by software, the handler routine executes a *RETI* (return from interrupt) instruction, which causes the old PC and SSW values to be restored from the supervisor stack, and the program picks up where it was interrupted.

Memory management

The physical main memory of most computers is organized as a set of consecutively numbered storage cells, each containing a byte of data. The location numbers associated with these storage cells are called real addresses, or physical addresses, and the set of all the real addresses is called the *real address space* of the computer. The real address space is thus determined by the actual hardware of the computer's memory system.

On the other hand, a program running on the computer generates a set of addresses as it executes, and this set is limited only by the maximum number of bits possible in an address (that is, the width of the program counter). This set of possible addresses is called the *virtual address space* (or sometimes the logical address space) of the computer.

Note that the virtual address space need not be the same size as the real address space; in fact the virtual space is usually much larger. For example, consider a 32-bit computer, such as the Clipper, with 4 million bytes of memory (a fairly typical configuration). A program on this computer can address more than 4 billion locations, making the virtual address space a thousand times larger than the physical memory. Here, we discuss three aspects of the Clipper's memory management scheme: memory architecture, address translation, and virtual memory. In addition, we describe how the CAMMU chip implements the memory management functions.

Memory architecture. The Clipper memory architecture is defined by the layouts of physical address space and virtual address space. Figure 13 is a diagram of the Clipper's memory architecture.

Physical address space. The maximum amount of physical memory that can usefully be connected to a computer is determined by the number of address lines in the system bus. For the Clipper this number is 32, so the Clipper's maximum physical memory and thus its maximum real address space is 4 gigabytes. Of course, most systems will not implement the maximum possible real address space; several megabytes is far more typical. We will however continue to refer to 32-bit real addresses, even though in a system with, say, 16 megabytes of physical memory only the low-order 24 bits of any real address will ever be nonzero (assuming physical memory begins at location 0 and has no gaps).

The Clipper puts additional information on the system bus along with the real address; one important item is a 3-bit quantity called the *system tag*. The tag is described in more detail below, but one of its intended purposes is relevant here, namely, to distinguish three separate physical address spaces:

- main memory space,
- boot ROM space, and
- I/O space.

Main memory space is the usual physical read/write memory. A boot ROM is a separate read-only memory containing initialization code that is automatically activated when the computer system is reset. It is useful to have these two be separate address spaces, so that the boot ROM can start at location 0 without either permanently taking up a chunk of the lowest addresses in main memory or requiring sophisticated circuitry to switch it in and out of main memory as required. Clipper provides this switching logic on chip, so the memory control logic need only decode the system tag and send addresses to main memory or to the boot ROM as appropriate.

The Clipper uses memory-mapped I/O; that is, it does not employ special I/O instructions. Instead the usual loads and stores that are employed to access memory are also used for input and output. The I/O devices on the system bus are responsible for recognizing which addresses are intended for them. Memory-mapped I/O is fairly common in microprocessors, but it usually requires dedicating a range of main memory addresses for I/O. In the Clipper system, I/O has its own, complete 4-gigabyte address space, which doesn't interfere at all with the complete 4-gigabyte main memory space. The I/O devices and memory control logic only have to decode the system tag to determine which of them is the intended recipient of an address.

Virtual address space. The virtual address space of the Clipper is very straightforward: It is a linear, unsegmented 4-gigabyte space. Just as the system tag on the bus effectively creates three separate real address spaces, so do the internal operating modes of the Clipper create different virtual address spaces. It is useful to think of the Clipper as having four virtual address spaces at any one time: *user instruction space, user data space, supervisor instruction space,* and *supervisor data space.* Programs running in user mode generate addresses in one of the two user spaces (just as they access the user registers), while supervisor-mode programs access the two supervisor spaces. As we shall see in the next section, the Clipper provides hardware mechanisms that support these four separate spaces.

Address translation. Address translation is the process that maps virtual address space onto real address space. *Mapping* is the address translation scheme that allows virtual addresses to be translated into arbitrary real addresses; it provides a kind of generalized relocation mechanism. Without mapping, multiprogramming is a very difficult and risky proposition.

For efficiency, mapping is usually done in blocks of addresses instead of independently for each virtual address. The most widely used systems are based on fixed-sized blocks called *pages*, which are usually some low multiple of 1K bytes in length. The Clipper uses a page-based mapping scheme with 4K-byte pages.

In the page-based mapping system, virtual address space is broken up into thousands of pages, each with the same size. Real address space is also broken up into pieces, called *page frames,* having this same size. The mapping operation associates a page frame in real space with each page of virtual space. Since mapping is done in units of a page, two addresses close to each other in the same virtual page will also be close in the same real page, but two contiguous virtual pages may not be contiguous in real memory.

The Clipper's virtual address space of 4 gigabytes is divided into 1 million pages, each with 4K bytes. The real address space is similarly divided into 4K-byte page frames. The 4K-byte size was chosen after careful study. It is consistent with the industry trend toward larger pages, and it helps performance in four ways:

- It provides a high hit rate for the TLB (the on-chip buffer of page addresses described later).
- It is an efficient unit of disk transfer, thus improving I/O performance.
- It allows a larger cache to be accessed concurrently with the address translation process—since fewer bits participate in address translation, more can be used to select a set in the cache.
- It permits two-level mapping; smaller pages would require more levels.

When mapping is enabled (by setting a bit in the system status word), the CAMMU translates virtual addresses generated by the CPU/FPU into real addresses. In much the same manner as in mainframes, the translation process is accomplished with a two-level hierarchy of *page tables* as seen in Figure 14.

Each process has its own collection of page tables that defines its virtual-to-real address map and thus its own address space. The base of the hierarchy is the *page table directory;* it is one page long and contains 1024 32-bit entries, each of which can point to a page table. Page tables themselves are also one page long and contain pointers to pages.

Each CAMMU contains two special *page directory origin* (PDO) registers. One points to the base of the page table directory of the supervisor-mode program (the operating system). The other points to the base of the page table directory for the currently executing process. Upon a process swap, the operating system can simply change the user PDO to obtain a new user address space. Since the Clipper offers separate CAMMUs for instructions and data, each with its own PDO, there are actually four memory maps, and thus four address spaces, active at any one time:

- supervisor instruction space,
- supervisor data space,
- user instruction space, and
- user data space.

Figure 15 shows in more detail how the address translation process works. The operating system has programmed the PDO register to point to the base of

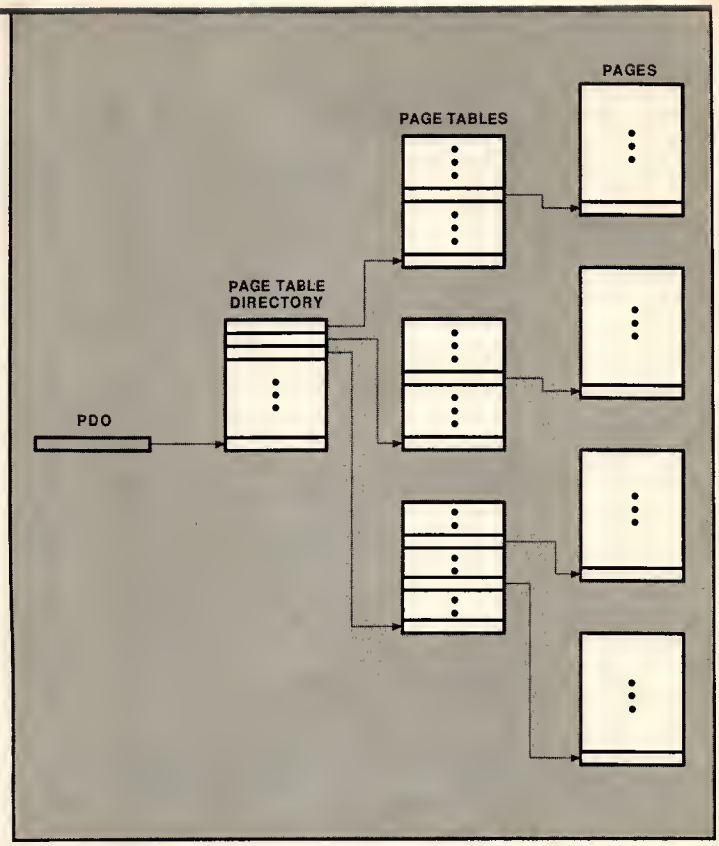


Figure 14. Two-level address translation.

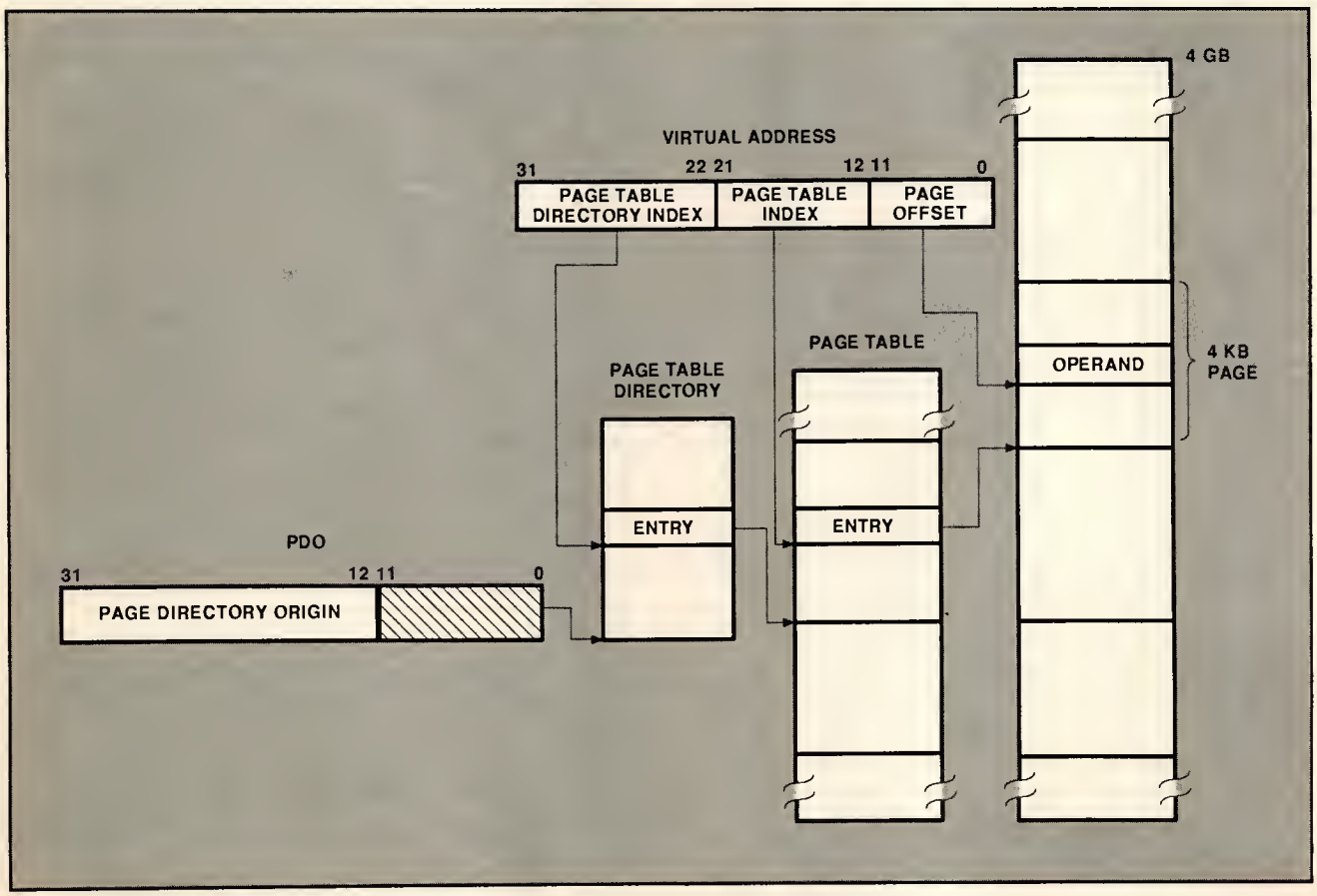


Figure 15. Clipper address translation.

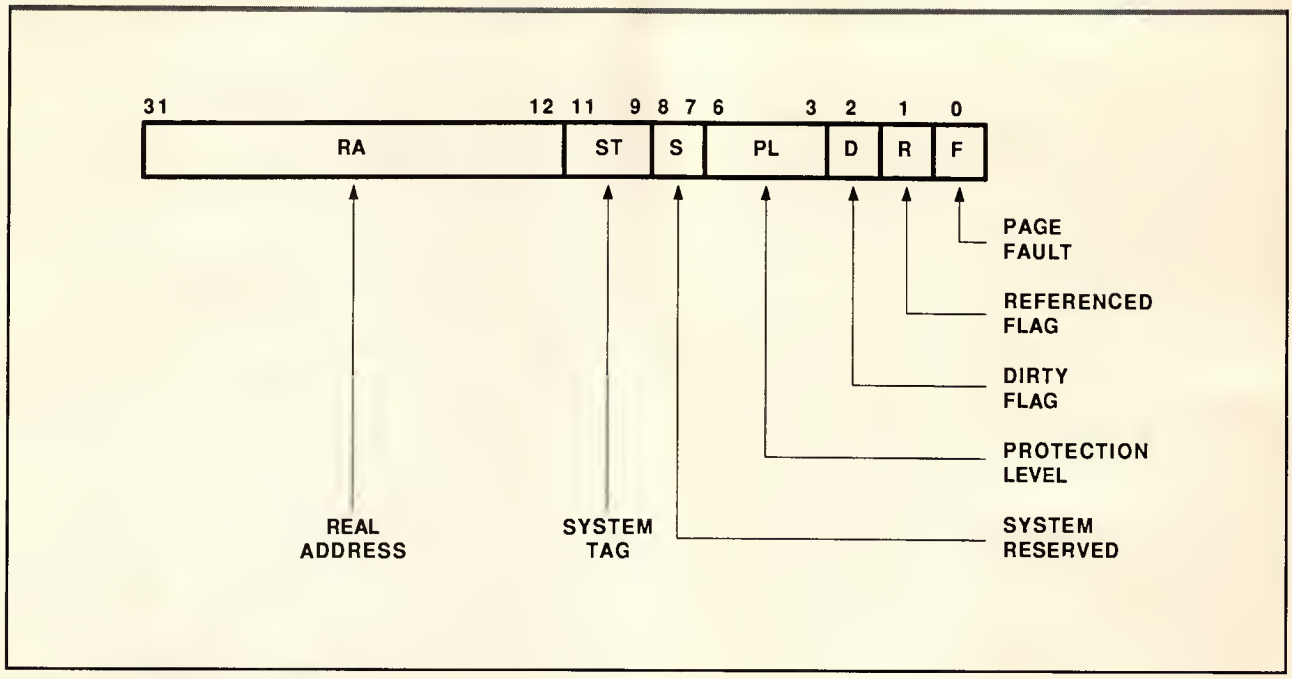


Figure 16. Page table entry.

the appropriate page table directory. When the CPU sends a virtual address to the CAMMU, the upper 10 bits of the virtual address are used as an index into the current page table directory. The selected entry contains the real address of a page table. The middle 10 bits of the virtual address are used as an index into this page table. The entry selected this time contains the real address of a page frame. Concatenating the lower 12 bits of the virtual address to the real address of the page frame produces the 32-bit real address of the operand.

To save the overhead of page table lookups every time an address is sent from the CPU, each CAMMU caches address translation information on 128 frequently used pages in a translation lookaside buffer. The TLB is searched concurrently with cache accesses in the CAMMU. Only when mapping data for the requested page is not in the TLB does an access to a memory-resident directory or page table have to be made. The Clipper TLB has a hit rate in excess of 99 percent.

As was mentioned above, the Clipper at any one time has four virtual address spaces (supervisor data and instruction, user data and instruction) and three real address spaces (main memory, boot ROM, and I/O). The PDOs in each CAMMU determine the maps that define the two supervisor and two user virtual spaces. The page table entries (see Figure 16) contain the system tags that define which of the three real address spaces is accessed. When a virtual address is translated, this tag value is put on the system bus along with the translated real address. The table entry also contains protection information about the page. For example, pages can be set to read only, execute only, or limit access to the supervisor. Other bits in the table entry are useful for virtual memory.

The lowest eight pages of the supervisor virtual address space are permanently mapped, via hardwired logic in the CAMMU. Figure 17 shows this mapping. Virtual pages 0-3 are translated into real pages 0-3 (main memory); virtual pages 4-5 are translated into real pages 0-1 (I/O); and virtual pages 6-7 are translated into real pages 0-1 (boot ROM). This permanent mapping provides several benefits. It makes the boot ROM immediately available on reset; it also makes some I/O available during initialization; and finally, it ensures that the lowest three pages of the supervisor's address space (which are in constant use, since they contain the exception vector table) are always translated rapidly.

Pages can be shared between processes by putting an entry for the same real page frame in page tables belonging to each process. The supervisor can access user pages by a similar mechanism—mapping one of its pages into a page frame that is also being used by a user program. The Clipper also provides a special mechanism that lets the supervisor use the user PDO for operand addresses; this facilitates rapid access by the supervisor to the entire user address space.

Virtual memory. Virtual memory is an operating system mechanism (supported by computer hardware) that allows large programs, or groups of programs, to circumvent the limitations on the amount of physical memory in a computer system by exploiting disk storage for some pages. In a virtual memory system it appears to the user that the entire virtual address space is available for access; but actually at any given moment only a few virtual pages are mapped into physical address space. The rest are stored on disk.

Whenever the processor generates a virtual address, the hardware checks to see if that address lies

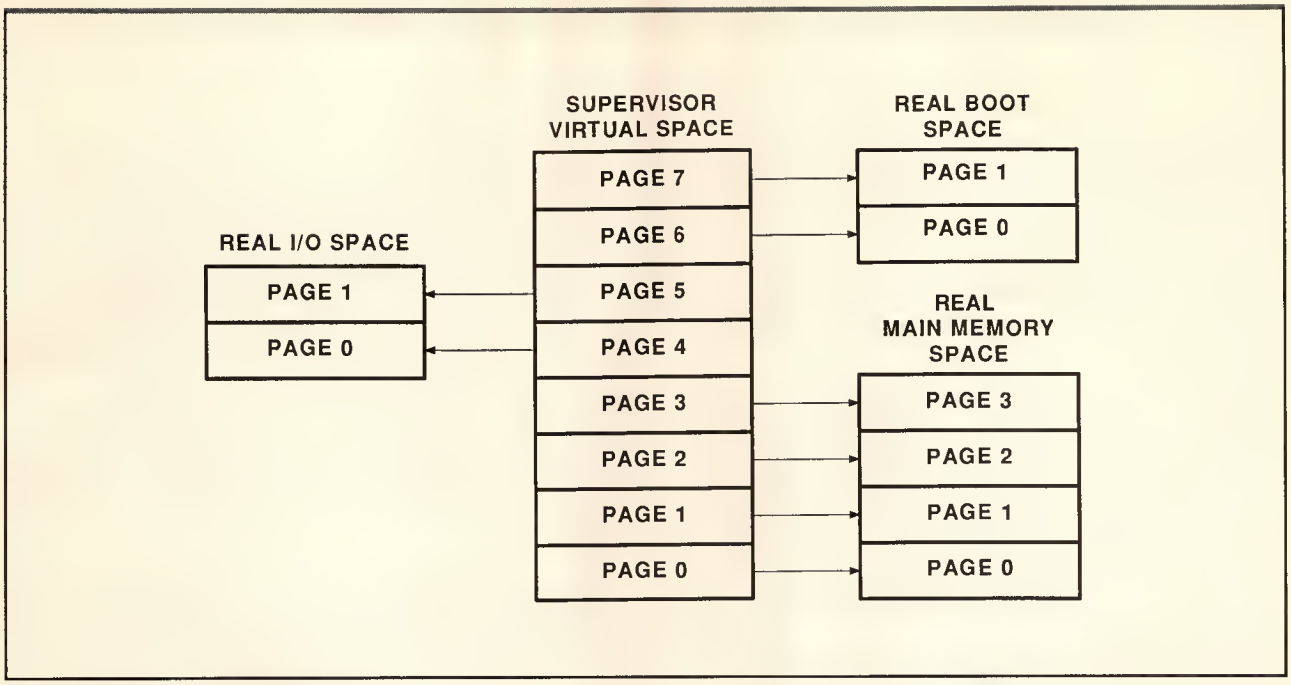


Figure 17. Hardwired mapping.

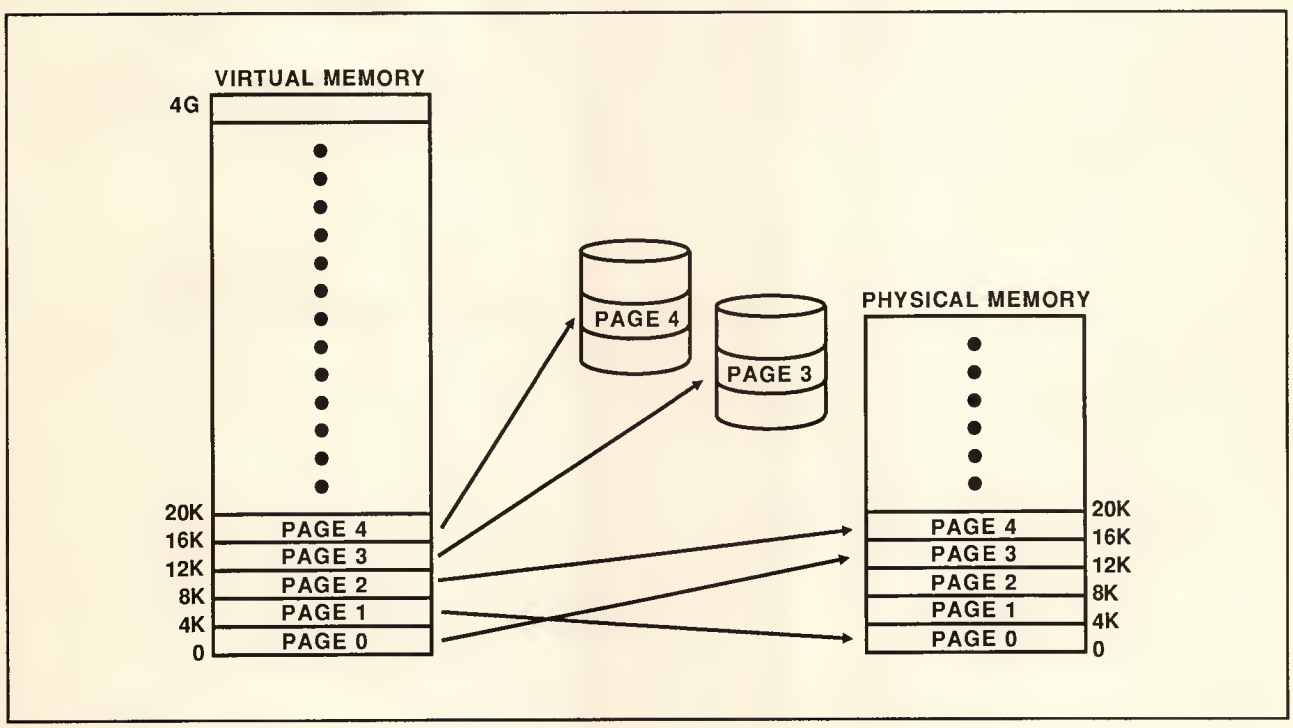


Figure 18. Demand paging.

in a page that is actually in memory. If it does, the address is translated normally. However, if the page is not in memory, an operation called a *page swap* is performed, and the operating system loads the missing page from disk. If this swap is performed rapidly enough and if missing pages are relatively infrequent, the virtual memory system performs nearly as well as one with far more memory, and at a fraction of the cost. This process, called *demand paging,* is diagrammed in Figure 18.

The Clipper architecture provides four key architectural features to support demand-paged virtual memory:

- a bit in the page table entry for each page that tells the CAMMU if a page is absent from main memory;
- a special processor trap, called a *page fault,* that can be activated by the CAMMU when a "not-present" page is accessed;

• the capability of aborting instruction execution when a page fault occurs and reexecuting or resuming the instruction after the operating system has loaded the missing page; and
• bits in the page table entries that facilitate the choice of the optimal page to be replaced when a newly swapped-in page must evict a page currently in memory.

As Figure 16 showed, a Clipper page table entry contains the page fault (F) bit, which is set if the corresponding page is absent from memory. The page fault exception is one of the 18 hardware trap conditions; when it is activated, the operating system can inspect the address that caused the fault (it is stored in a special CAMMU register) and then swap in the missing page. All Clipper instructions are fully restartable.

Initially, all pages are on disk, and memory is totally free. But as the program executes and generates addresses that cause page faults, more and more pages come into memory. Eventually memory becomes full, and newly swapped-in pages must replace pages in memory. At this point the operating system attempts to replace pages that it predicts are least likely to be referenced in the near future. One popular algorithm for making this prediction selects the page that was least recently accessed. The referenced (R) bit in the page table entry supports this algorithm. This bit is set automatically whenever the corresponding page is accessed. By periodically examining and clearing the bit, the operating system identifies pages that have not been used recently.

Once the operating system has selected a page to evict, it must decide whether to write the page back to disk or simply discard it. The dirty (D) bit facilitates this decision. This bit is automatically set whenever the corresponding page is modified. Clearly, if this bit has not been set, the data on disk already matches the page, and there is no need to write the page back.

Clipper MMU implementation. The CAMMU contains the cache mechanism and the memory management unit. The cache was described earlier and pictured in Figures 2-5; here we discuss the MMU. Basically the MMU contains two parts—the dynamic translation unit, DTU, and the TLB. Figure 19 shows a diagram of the CAMMU; the right side comprises the MMU.

The TLB is a two-way set-associative cache that is used by the MMU for fast address translation requiring no access to main memory. It consists of 64 sets

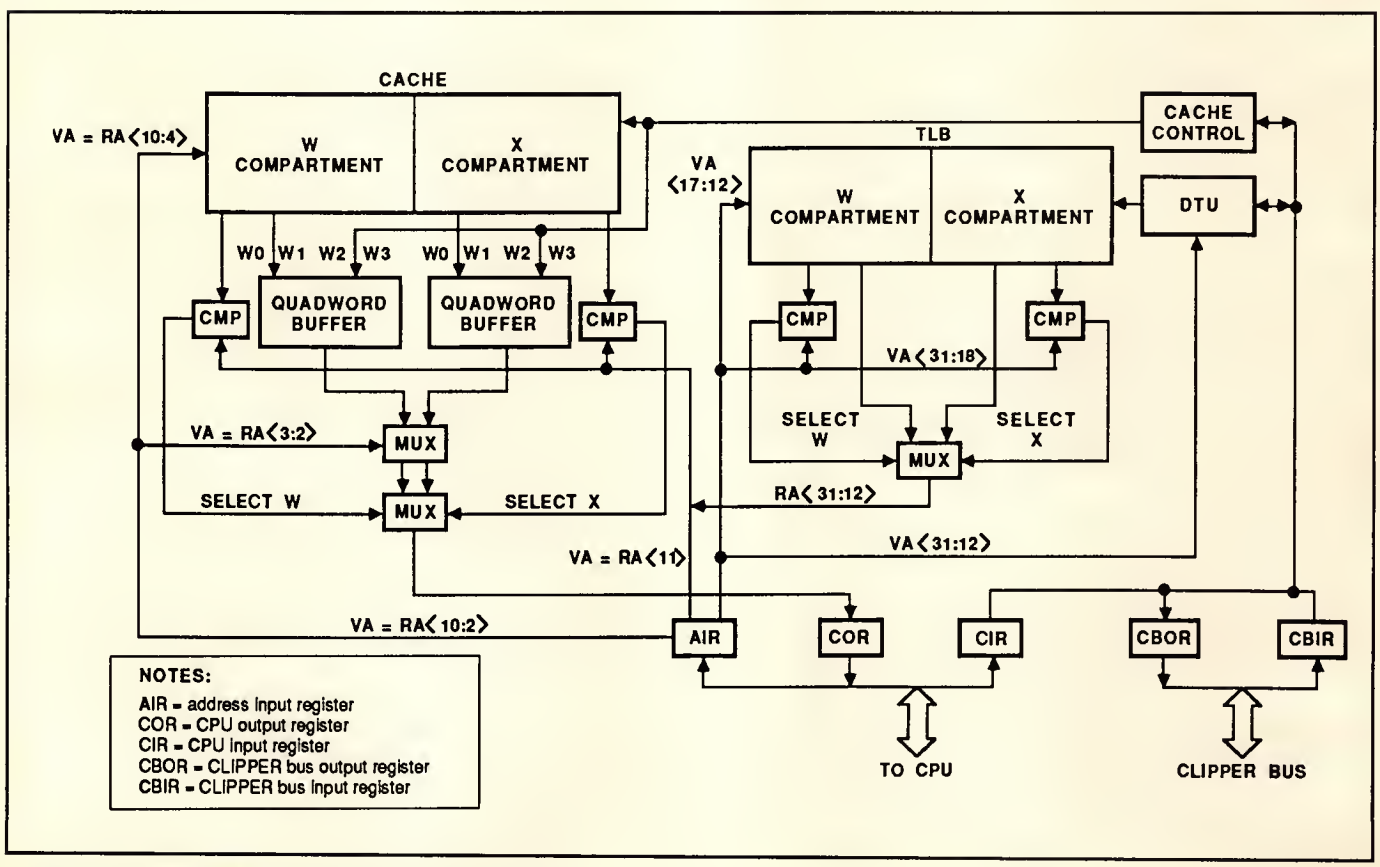


Figure 19. Block diagram of the Clipper CAMMU.

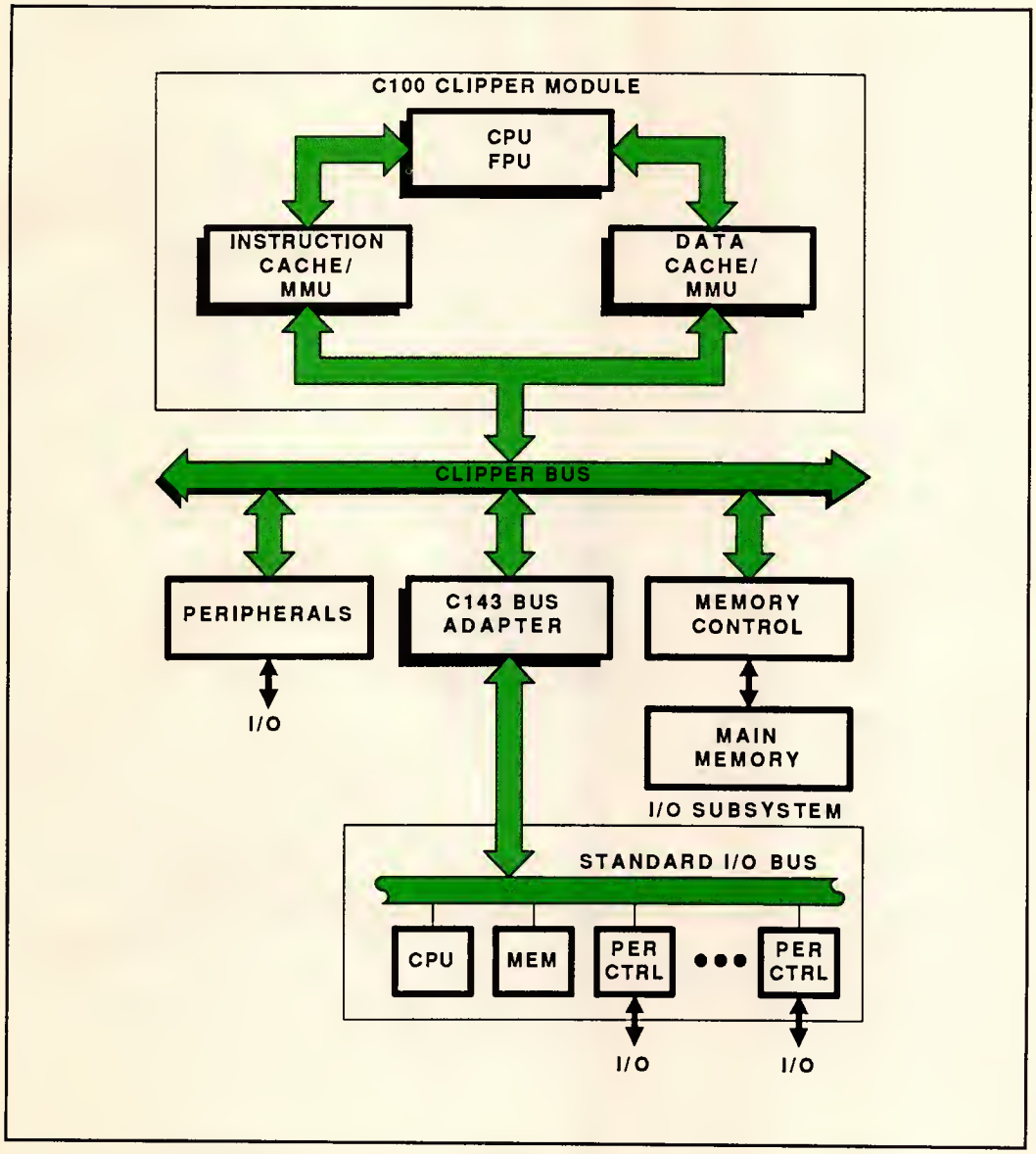


Figure 20. Typical Clipper system.

of lines, with each set containing a W and an X compartment line.

The TLB operates concurrently with and in a very similar fashion to the cache for each virtual address presented to the CAMMU. Bits 12-17 of the virtual address are used to select a TLB line set at the same time bits 4-10 are selecting a cache line set. Then bits 18-31 of the virtual address are compared with the virtual address field of both lines in the set. If a match occurs, we have a TLB hit, and the corresponding real address field is used by the cache in its comparisons. If there is no match, we have a TLB miss, and the DTU attempts address translation using page tables in main memory. Once the DTU has completed this process, the CAMMU updates the TLB with this latest translation, replacing an existing line if necessary. Then the cache access occurs again.

System architecture

Thus far we have focused primarily on the internal and instruction set architectures of the Clipper. This final section is devoted to the remaining features of the architecture of Clipper-based systems. We use the expression *system architecture* as a shorthand expression for these features.

Clipper-based systems are modular. They consist of the Clipper module itself and two main subsystems, memory and I/O. Connecting the subsystems and the Clipper module is a bus called the Clipper Bus. Figure 20 shows a high-level diagram of a complete Clipper computer system.

The Clipper module is a 3 × 4.5-inch, multilayer printed circuit board holding the Clipper CPU/FPU chip, the two CAMMU chips, and a clock chip plus a

66.66-MHz crystal. The components are surface mounted. A 96-pin DIN standard connector is also mounted on the edge of the module; this connector is used to interface the module to the rest of the system.

Seventy-three of the 96 pins in the DIN connector are utilized for signals; these lines define the Clipper module interface. The CMI includes 32 address/data lines, eight interrupt vector lines, six lines defining the type of operation in progress (read, write, length of transfer), three lines defining the system tag, and a number of control lines, including interrupt control and bus arbitration for systems with multiple bus masters.

The clock chip generates two clock signals, MCLK and BCLK. MCLK is the internal Clipper master clock, used to drive the CPU/FPU, the CAMMUs, and associated logic. The frequency of MCLK is half the frequency of the module's crystal, that is, 33.3 MHz. BCLK is a signal on the CMI; it is the system clock, used for timing on the system bus. BCLK's frequency is one fourth the crystal frequency, or 16.6 MHz.

The Clipper processor was designed to be, as much as was practically possible, a supercomputer architecture on a chip. The compromises that were required in the course of its development (three chips instead of one and 4K-byte caches) were minor compared with what has been achieved. The Clipper CPU includes separate integer and floating-point units, a first for microprocessors. The CAMMU's use of two-way set-associative caches and bus watch mechanisms is also new for microprocessor architecture. The result is a microprocessor whose performance compares favorably with much larger mainframes and super-minicomputers.

Acknowledgments

I thank several people at Fairchild Advanced Processor Division for their generous assistance in the preparation of this article, especially Howard Sachs and Walt Hollingsworth, the chief architects, and Gary Baum and David Neff, who offered many valuable suggestions for improvements.

Alexia Gilmore and Erin Farquhar also provided welcome assistance.

References

1. R. Mateosian, "System Considerations in the NS32032 Design," *Proc. NCC,* 1984, pp. 77-81.
2. A.J. Smith, "Line (Block) Size Selection in CPU Cache Memories," June 1985, report UCB/CSD85/239 (Computer Sciences Division, Univ. of California, Berkeley).
3. C. Alexander, et al., Texas Instruments and Rice Univ. preprint of *Cache Memory Performance in a Unix Environment,* 1986, submitted for publication.
4. M. Hill and A. Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proc. 11th Int'l. Symp. Comp. Arch.,* June 1984.
5. D. Patterson, "Reduced Instruction Set Computer," *Comm. ACM,* Assoc. of Computing Machinery, New York, Jan. 1985.
6. V. Milutinovic, *High-Level Language Computer Architectures*, Computer Science Press, Rockville, MD, 1987.

Colin B. Hunter is president of Hunter Systems, Inc., Mountain View, California, the developer of XDOS, a software product that lets IBM PC programs run on computers not based on the 8086 architecture. Previously, he was the president of Hunter & Ready, Inc., the developers of the VRTX real-time operating system for microprocessors. He has written two books on computer architectures and contributed several articles on the subject.

Hunter has a BS in mathematics from Stanford University and is a member of the IEEE and the American Mathematical Society.

Questions about this article can be addressed to Hunter at Hunter Systems, Inc., 444 Castro Street, Mountain View, CA 94041.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 150 Medium 151 Low 152

Jim Blinn's Corner

the first edition in the August '87 issue of CG&A

with the man who created the Mechanical Universe

Just call (714) 821-8380

FEATURE

SYSTEM CONSIDERATIONS IN THE DESIGN OF THE AM29000

Mike Johnson
Advanced Micro Devices

The Am29000 blends the flexibility of microprogrammed processors with standard programming languages and operating systems in a second-generation RISC.

The development of the Am29000 Streamlined Instruction Processor began in mid-1984 within Advanced Micro Devices' programmable processors product planning group. Our traditional markets provided opportunities for a processor family which could be programmed using existing, standard software tools. Previous AMD bit-slice and functional-slice processors, since they encouraged many varied, nonstandard instruction sets—with software development at the microcode-assembler level—restricted our ability to capitalize on state-of-the-art software technology.

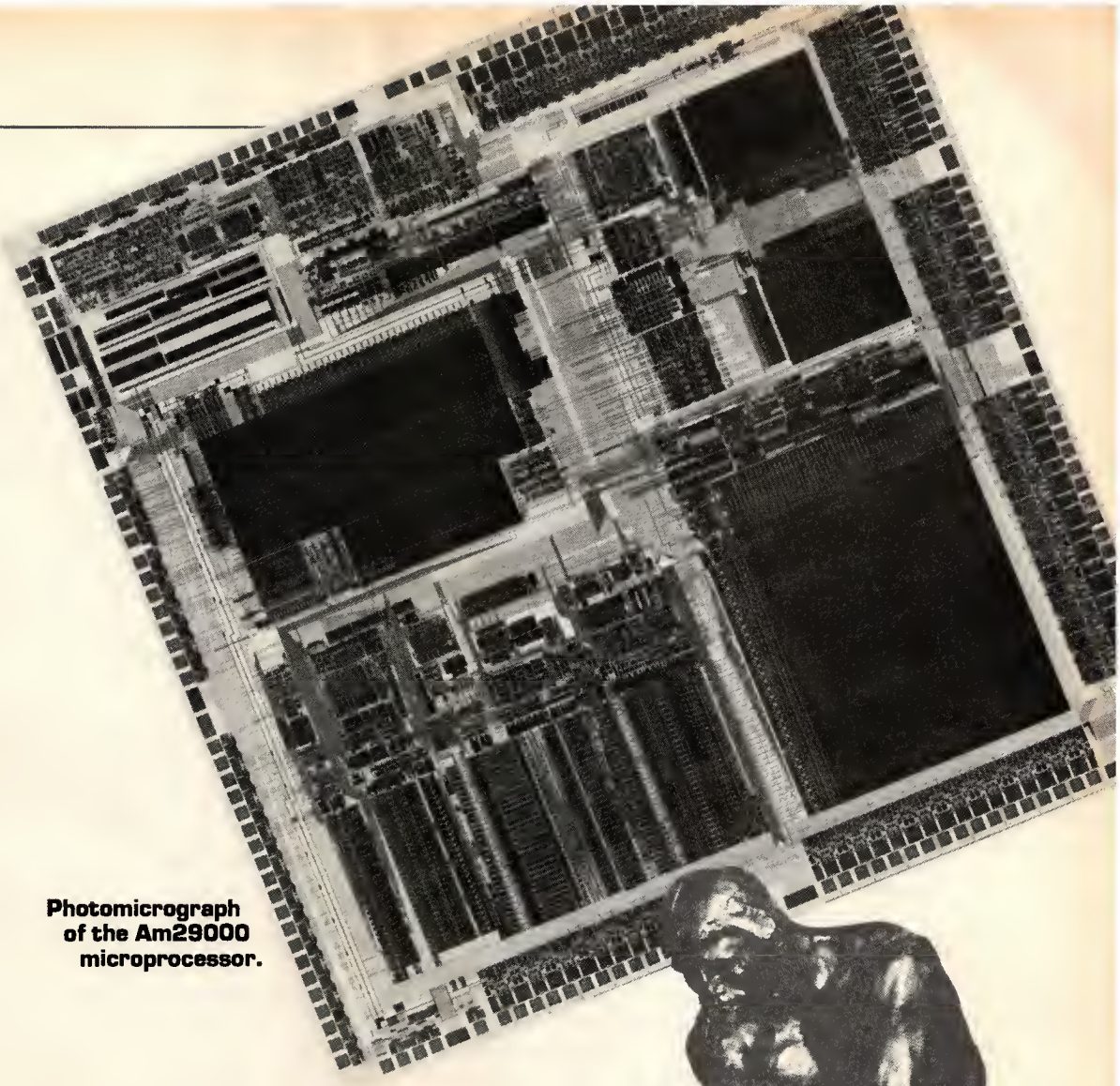
The philosophy behind our processor families has been maintained in the Am29000, but at a higher level. Instead of providing performance and flexibility at the microcode, instruction-set, and chip-interconnection levels, the Am29000 provides performance and flexibility at the compiler, operating-system, and hardware-environment levels. It shares a heritage with previous AMD processors, but with the added advantages of standard programming languages and operating systems.

Here, I explain some of the rationale behind the Am29000 product definition. I especially concentrate on system considerations, since the Am29000 is designed to provide performance in a wide variety of applications.

Design philosophy

Over the past few years, the technical press has featured diverse coverage on RISC architectures. The acronym RISC has stood for reduced instruction set computer, reusable information storage computer, relegate im-

0272-1732/87/0800-0028$01.00 © 1987 IEEE



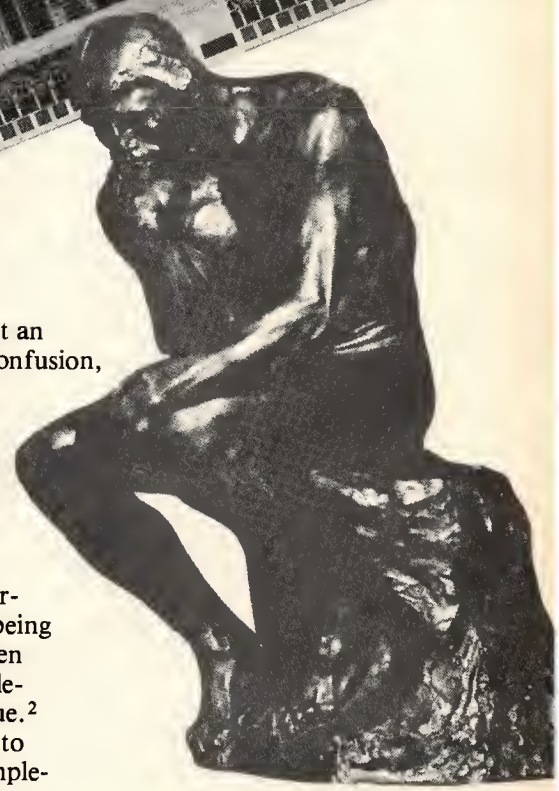
Photomicrograph of the Am29000 microprocessor.

portant stuff to compiler, or whatever else the reader might fancy. Unfortunately, RISC also tends to be a tag line for any 32-bit processor without an established market introduced since 1982. This, in my view, has caused much confusion, and has masked the essential ideas behind "the RISC philosophy."

RISC fundamentals. A complex hierarchy can be designed for maximum performance if each level of the hierarchy performs only those functions which it does efficiently. In modern software systems, the hierarchy includes high-level-language (HLL) applications, an optimizing compiler, and an instruction set interpreted by a processor.

As with many other engineering endeavors, the natural evolution of computer-system architectures has been "bottom up,"[1] with compilers and applications being built on top of existing instruction sets. The design of new architectures has often followed the line of reasoning that a system can be made more efficient by implementing high-level software functions in hardware. However, this is often untrue.[2]

The key idea behind a RISC is that a processor instruction set should not try to duplicate the function of an HLL compiler. Rather, it should concentrate on implementing those functions which cannot be implemented efficiently by an HLL compiler, and leave the compiler to decide on the use of these functions.[3]



"The Thinker" Auguste Rodin 1880.

Beyond the reduced instruction set. RISC principles have emphasized the simplification of the semantics of a processor's instruction set, so that this instruction set can interface more easily to an optimizing compiler.[4] However, a processor not only executes compiled programs, but also must interface to an operating system and hardware components. Processor operations can be simplified with respect to these other interfaces as well, with a resulting improvement in system performance.

When processor operations are simplified, the emphasis can shift to ensuring that these operations attain maximum performance. Achieving maximum performance is relatively simple, in a sense, since the scope of processor operations is intentionally limited to those which the processor can efficiently perform. Simplicity has many benefits—mainly that the overall system can have high performance rather than be limited by processor capability.

The processor's contribution to performance. Within the scope of a processor design, there are essentially three ways to boost performance:

- Reduce the number of operations performed,
- Reduce the cycle time, and
- Reduce the number of cycles for each operation.

Processor architecture or implementation does not have much effect on the first two items, except that an inappropriate instruction set can increase the number of operations required, and an inappropriate design can increase the cycle time. However, when a processor architecture is "reasonable" in these respects, reductions in these areas are essentially controlled by factors outside of the processor.

Compiler and software applications can most effectively optimize the number of operations, and the implementation technology (of both the processor and other hardware components) can most effectively optimize the cycle time. The bulk of a processor design, then, should concentrate on achieving an instruction-execution rate which is near single-cycle execution. One of the most satisfying aspects of RISC architectures is that they can simultaneously satisfy the requirements of compiler optimization, cycle time, and single-cycle instruction execution.

Pipeline efficiency. Single-cycle instruction execution at a reasonable cycle time requires that processor operations be pipelined. However, a pipeline cannot always work efficiently. Pipeline inefficiencies caused by register loads and successful branches can be especially difficult to deal with, since they include off-chip memory paths.

Software techniques can reduce the effects of the external memory delay. For example, the processor can implement overlapped loads and delayed branches, so that external accesses can be performed concurrently with other instructions. These mechanisms can be very effective if only a small amount of delay is present in the memory path. However, they are difficult to generalize into techniques which work effectively in high-delay situations.

Another mechanism for dealing with memory delay is to introduce a large number of general-purpose registers into the architecture. These registers can significantly reduce the number of external data accesses required in most applications. If the registers are allocated properly, many references which would otherwise be directed to an external memory can be directed instead to these registers.

Memory delays. In spite of the above observations, the most important factor in pipeline efficiency is the delay of the memory path. Not only is this delay a result of the memory access time but also a result of the protocols involved in a memory access. In particular, pipelining the protocol to reduce the logic delays involved does not decrease the overall delay, but, in most cases, increases it.

Some processor designs introduce delays because of contention for resources between instruction and data accesses. This contention is either for shared buses, shared memories, or, more subtly, some other shared resource such as a memory management unit.

The most practical solution to the contention between instruction and data accesses is a memory interface which can handle two accesses simultaneously. This is accomplished by separate buses and memories (or memory ports), by a memory interface which runs at twice the processor's frequency, or by some combination of these techniques. However, any solution which requires doubling the frequency of operation for some system component (especially a bus) has a serious inherent performance limitation.

Achieving true single-cycle instruction execution. The problems briefly described above motivate the following conclusion: The simple instructions of a RISC enable a single-cycle instruction-execution rate, but simple instructions alone do not guarantee that this rate will be achieved.

So-called complex-instruction processors inherently limit the instruction-execution rate to a relatively high number of cycles per instruction, because of the number of operations performed by the average instruction. In a sense, this reduces the burden on the rest of the system. However, it also means that the processor cannot take maximum advantage of a high-performance system design, because of the instruction-execution bottleneck.

Though RISCs eliminate the instruction-execution bottleneck, RISC principles do not fully address the problems associated with keeping the processor supplied with instructions and data at the maximum rate. In this respect—keeping the processor operating at full speed in the presence of register loads, stores, branches, and memory delays—the design of a RISC is complex. There are many interacting effects, and it is tempting to make compromises in one part of the system because of a deficiency in some other part.

However, any compromise from the ideal of single-cycle execution is likely to represent a considerable impact on system performance. For example, an increase of a "mere" 0.2 cycles/instruction represents a 20-percent decrease from ideal perfor-

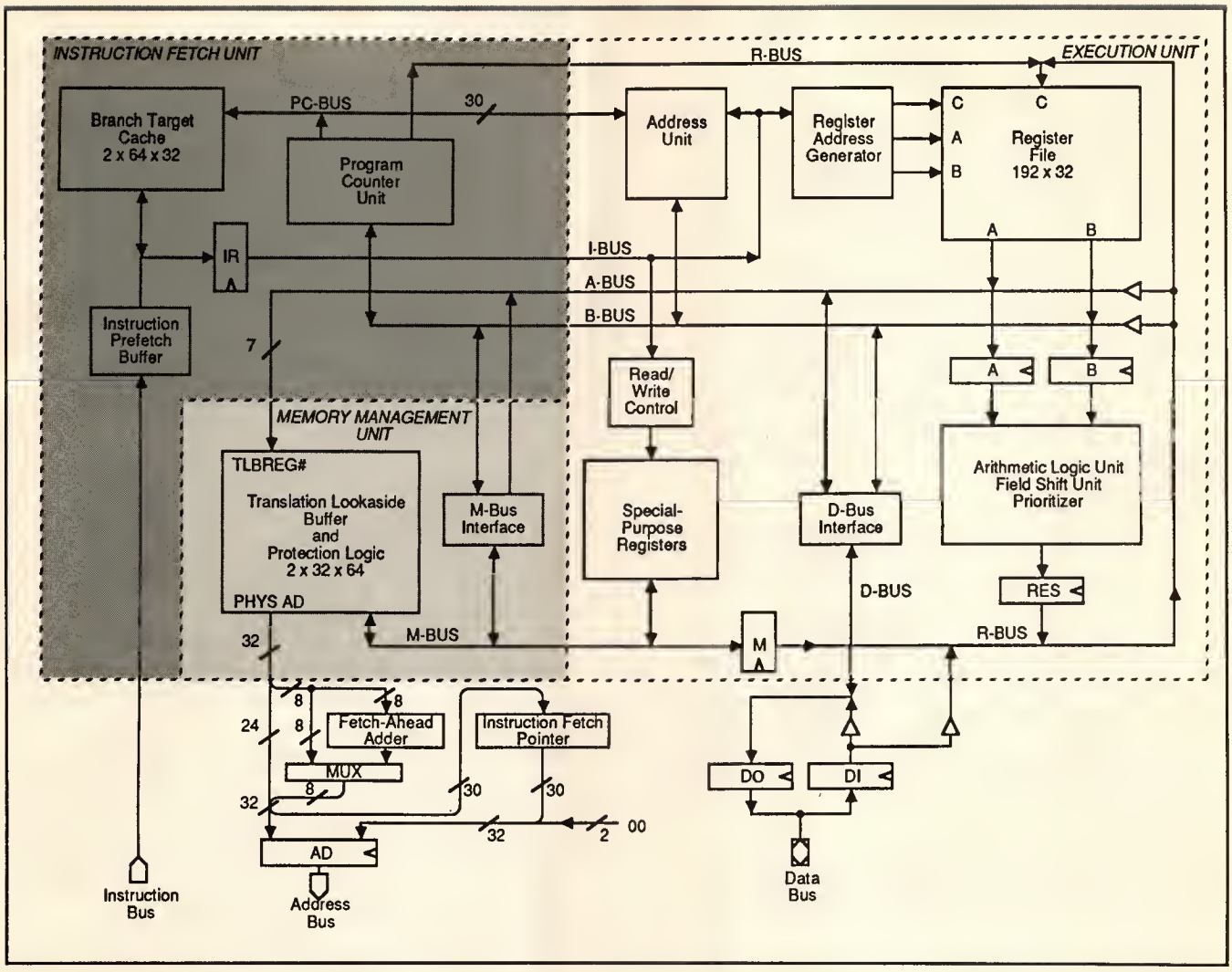


Figure 1. Am29000 dataflow.

mance. When considering a particular processor feature, the best way to avoid such compromises is to abstract all other parts of the design by assuming that they achieve the maximum possible performance, regardless of the current state of the design. In this manner, the entire design can be brought near the ideal of single-cycle instruction execution.

Am29000 definition

Several design trade-offs and decisions were made during the product definition of the Am29000. Figure 1 shows a dataflow diagram of the Am29000, and can be a useful reference during some of the following discussion.

The objective of the Am29000 is to take advantage of simplicity, while performing functions required by a wide variety of applications using the same basic resources. The result is performance, flexibility, and reduced system cost.

A large portion of the Am29000 architecture is devoted to allowing true single-cycle execution. In real applications, the Am29000 can execute instructions at the peak execution rate of 25 million native instructions per second at a 25-MHz operating frequency.

Register file. One of the first design decisions was that the Am29000 should incorporate a large number of general-purpose registers. This follows the normal RISC pattern, except that the Am29000 makes 192 registers available to any given instruction, which far exceeds the typical numbers of 16, 32, or 64 registers for other RISCs.

Am29000 register organization. The Am29000 register file includes 64 fixed-address registers, called global registers, and 128 variable-address registers, called local registers. These are depicted in Figure 2.

The global registers are allocated statically by the compiler or system designer. They may be used, for example, to hold temporary variables for expression evaluation, or for operating-system values such as page-table base addresses. Depending on the sophistication of a compiler's register-allocation mechanism,[5] they may also be used to contain a number of statically allocated variables (for example, C global variables).

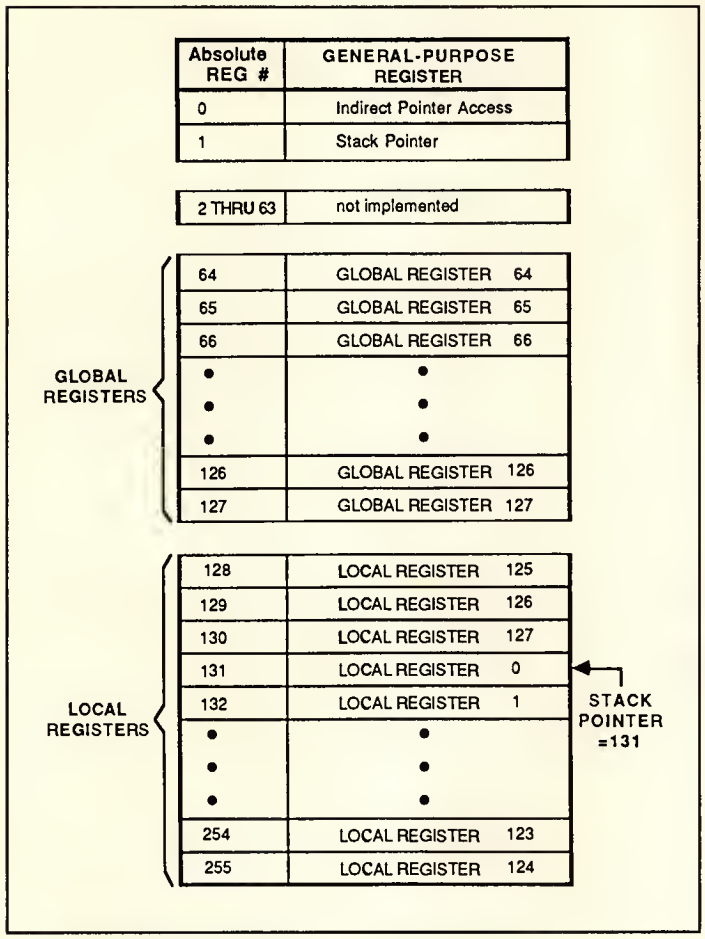


Figure 2. Register organization.

The local registers are identical to the global registers except in one respect. Addresses for local registers are relative to a stack pointer, as indicated in Figure 2 (the value 131 for the stack pointer is used in Figure 2 only as an example). The absolute register-number for Local Register 0 is given by bits 9-2 of the 32-bit stack pointer, and all other local registers are addressed relative to Local Register 0. The addresses of local registers are computed modulo 128.

Runtime stack cache. The relative addressing applied to the local registers allows these registers to be easily allocated across procedure calls. Normal register-allocation mechanisms used by compilers[6] are effective only for registers allocated within a procedure. When a new procedure is called in the Am29000, a fresh set of registers is allocated to the new procedure simply by adjusting the stack pointer. Since the called procedure can access the registers of the calling procedure, parameter-passing is also easily accomplished.[7] In effect, the most recently used portion of the runtime stack (or, at least, the portion of the runtime stack containing scalar values) is cached within the local registers.[8]

With this register-allocation scheme, it is possible that no local registers will be left for allocation to the new procedure when it is called. This condition, called *overflow*, is detected during procedure calls by software which compares the value of the stack pointer, after adjustment, with a bounding value kept in a global register. Overflow causes registers to be *spilled* onto a runtime stack in memory. This creates a corresponding condition, called *underflow*, on procedure return. Underflow occurs when the values required by a procedure were previously spilled onto the runtime stack, and must be reloaded.

Overflow and underflow are detected by software in the Am29000, using instructions for runtime address checking (such as computed array-index checking). It would have been possible to define some sort of hardware bounds checking, but the resulting performance improvement would have been negligible, with a significant loss in generality.

The overhead for allocating local registers and checking for overflow and underflow is two cycles on procedure call and three cycles on return. During the design, we thought that the total overhead would be six cycles. A different scheme was invented well after completed chip design, reducing the overhead to five cycles. Furthermore, we realized that leaf procedures (which do not call any other procedure) required no overhead for stack-cache management. A compiler can easily detect a leaf procedure and map its activation record to the global registers. These late improvements in the procedure-linkage mechanisms confirm the wisdom of leaving the stack management to software.

The benefit of caching the runtime stack in local registers is that overflow and underflow occur infrequently with respect to procedure calls, since the size of the runtime stack does not change very much over long program intervals. (Note that overflow and underflow are caused by a change in size of the runtime stack beyond that which can be contained in the local registers.) Simulations of the Am29000 show that the procedure calls requiring spilling and filling of local registers are typically one percent of all calls. This is in contrast to fixed-address registers, which can require register loads and stores on nearly every procedure call.[9]

Organizing the local registers as a stack cache is more effective than devoting the same amount of memory to an on-chip data cache. First, the register file has three ports, allowing two operand reads and a single result write in one cycle, while a data cache would have a single port. Second, the registers are accessed during instruction decode, so that the access time has no effect on performance (that is, access time is perfectly overlapped). Finally, a data cache would require the execution of explicit load and store instructions to read and write data, which is not the case with the register file.

Register banking. In the early stages of Am29000 definition, the organization of the general-purpose registers was essentially as just described. However, as the design progressed, it became apparent that the general-purpose registers were really more than just registers; they were, in effect, a small segment of RAM.

There was a recurring need to reserve certain global registers for the operating system. For example, the interrupt and memory-management functions were defined to allow as much flexibility as possible. However, this created the need to keep certain values (such as interrupt stack pointers, segment numbers, and page-table base addresses) in global registers.

Furthermore, with a different runtime organization, the register file could yield a significant boost to response time in a real-time environment. In the stack-cache configuration, most general-purpose registers are allocated to one process or task. On a task switch, saving all registers requires about 8 μs. However, if the registers could be partitioned among many tasks, a task switch could occur in less than 700 ns.

To address the system-integrity concerns raised by the above issues, we added an extremely simple register-protection mechanism. Register protection was defined to operate on banks of 16 registers (see Table 1). Sixteen protection bits are sufficient to specify the protection status for the 256-register address space, and various combinations of protection bits allow any desired partition of register banks. Protection is based on the supervisor/user modes, to closely match other processor protection mechanisms. With the basic supervisor/user protection, additional levels can be implemented by policies within supervisor-mode programs.

Table 1.
Register banking.

Register bank protect register bit	Absolute register numbers	General-purpose registers
0	2 thru 15	Bank 0 (unimplemented)
1	16 thru 31	Bank 1 (unimplemented)
2	32 thru 47	Bank 2 (unimplemented)
3	48 thru 63	Bank 3 (unimplemented)
4	64 thru 79	Bank 4
5	80 thru 95	Bank 5
6	96 thru 111	Bank 6
7	112 thru 127	Bank 7
8	128 thru 143	Bank 8
9	144 thru 159	Bank 9
10	160 thru 175	Bank 10
11	176 thru 191	Bank 11
12	192 thru 207	Bank 12
13	208 thru 223	Bank 13
14	224 thru 239	Bank 14
15	240 thru 255	Bank 15

Instruction set. The first decision made for the Am29000 instruction set was that all instructions would be 32 bits in length. Supporting variable-length instructions, though yielding better memory efficiency, reduces performance. Variable-length instructions are more difficult to decode and pipeline than fixed-length instructions, and are a major source of obscure architectural bugs, especially with respect to page boundaries in a demand-paged environment.

When all instructions are fixed at 32 bits in length, the definition of a RISC instruction set becomes, to some extent, a study in how to waste 32 bits of instruction. Though this comment may sound flippant, a fundamental trade-off is made, in all areas of computer science, between storage requirements and execution time. Normally, this trade-off is emphasized in programming, in the context of algorithm and data-structure design. However, it applies equally well in instruction-set design. Large, fixed-length instructions encode programs less efficiently (in terms of memory used) than small, variable-length instructions, but they can be processed much faster. Given that memory costs are dominated by one-time (and decreasing) costs, while execution-time costs are recurring (and generally increasing), the shift towards an instruction set which is less memory-efficient and more performance-efficient is justifiable.

RISC instruction sets with a fixed, 32-bit instruction length tend to emphasize large immediate constants.[4,10] The Am29000 instruction set, in contrast, places much less emphasis on large constants, and much more emphasis on register addressability. The instruction set widely supports only 8-bit immediate data, but allows any three of the 192 general-purpose registers to be addressed by a given instruction. Larger instruction constants are supported, but require one or two cycles and the use of a register. This instruction organization follows from the observation that large constants are not as useful as the ability to address a large number of general-purpose registers, since large constants are not frequently used.

Although compiler considerations dominated the Am29000 instruction-set definition, the desire was to provide efficiency in the widest anticipated range of applications and programming environments. This led to the definition of a small number of instructions which would be frequently used only in certain applications, and which could be expressed only in assembly language. These are shown in Figures 3 through 5.

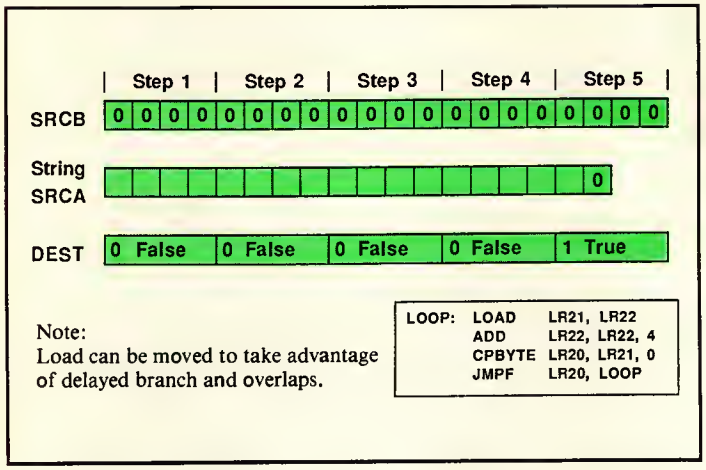


Figure 3. Compare Bytes instruction.

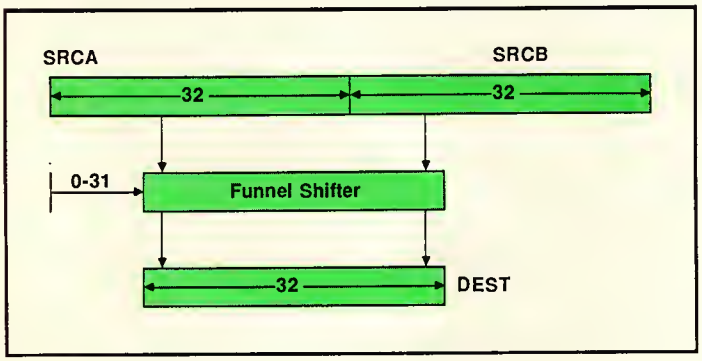


Figure 4. Extract instruction.

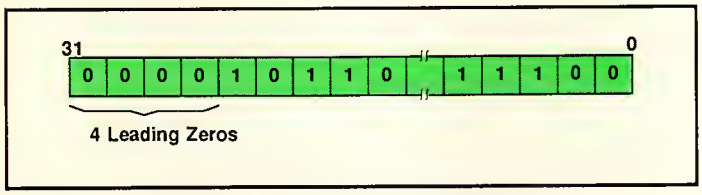


Figure 5. Count Leading Zeros in LR20; result in LR21.

In the Am29000, the Compare Bytes instruction (Figure 3) has a dramatic effect on the performance of string operations, because it allows many string operations to be performed in units of 32-bit words, rather than bytes. The Extract instruction (Figure 4) is useful in the movement of byte-aligned strings, and also aids raster operations in graphics systems. The Count Leading Zeros instruction (Figure 5) is useful for floating-point emulation and interrupt prioritization.

Data accesses. The Am29000 has a load/store architecture for external data references. All external data transfers occur to and from general-purpose registers, as the explicit result of load and store instructions. This approach is used by all RISCs, and has the advantage of being efficient and easy to implement, especially when considering the need to restart accesses in a demand-paged environment.

In contrast to most processors, the Am29000 performs no address computation. All addresses are register-indirect, and are contained in (or destined for) a general-purpose register at the beginning of an access. The omission of implicit address computations was motivated by the desire to reduce the delay for external data accesses to an absolute minimum, and to completely expose all address computations to compiler optimizations.

The most frequently used addressing mode in any system is a simple base-plus-offset reference to the runtime stack in external memory. Within a procedure, the address of the top of the runtime stack is typically contained in a processor register, and procedure variables are accessed via their offsets from this address. However, in the Am29000, a base-plus-offset mechanism for accessing the runtime stack is provided in the addressing of local registers. This reduces the need for such an addressing mechanism within the context of loads and stores.

What remains in the Am29000, then, are the address computations required to access external data other than scalars on the runtime stack. These address computations are often performed for references to structures such as arrays or records, and compiler optimizations can typically be very effective in reducing the amount of time spent on such computations.

Since load and store addresses are register-indirect in the Am29000, addresses are known at the end of the register-fetch cycle. This allows the access to begin early, and reduces the minimum access time by one cycle. The Am29000 performs address translation during the time that most processors would be performing address computation.

By not forcing address computation, the Am29000 provides an optimization opportunity to the compiler. For example, if the compiler is able to eliminate an address computation by common subexpression elimination, the Am29000 provides a corresponding reduction in the access time. This is yet another example of an improvement in performance which results from simplifying processor operation.

Overlapped loads and stores. Overlapped loads and stores allow the Am29000 to continue instruction execution while a load or store instruction is in progress. This capability allows an optimizing compiler to schedule loads and stores in the instruction stream so that the apparent delay is reduced or eliminated. If the Am29000 detects a dependency on data from an uncompleted load, it suspends execution until the load completes, eliminating the burden on the compiler to schedule all loads.[11]

In order to simplify the implementation of overlapped loads and stores, only one overlapped access is allowed at any given time. This approach reduces the amount of hardware required to track data dependencies and to restart faulting accesses in a demand-paged environment.

In a demand-paged environment, overlapped loads and stores must be reliably restarted in case of an exception (such as a page fault). The solution in the Am29000 is to buffer all information required to restart the overlapped access. If required, the access can be restarted using the buffered information, rather than by reexecuting the original load or store instruction. This restart is accomplished automatically during interrupt return.

Data forwarding. When data is returned to the Am29000 on the completion of a load, it can be used immediately if required for instruction execution. The logic used to route the data immediately to the appropriate execution unit is also used to solve a problem inherent in overlapped loads.

At the completion of an overlapped load, data must be written into the register file, but, for true single-cycle execution, the writing of data cannot take cycles away from instruction execution. A second write port for this purpose would have been prohibitive, given the size of the Am29000 register file. The solution in the Am29000 is to defer the register file write on the completion of the load. The load data is simply buffered in a latch when the load completes, and remains buffered until the next load or store is executed.

While this data is buffered, the logic which detects the pipeline dependency on load data is still active. However, instead of indicating that the processor pipeline must be held, it indicates that the required data is held by the buffer, instead of by the register file. In effect, the pipeline-dependency comparison acts to map the register file location to the data buffer. The buffer is not needed until the next load is executed, and the next load opens up an opportunity for the data from the first load to be written into the register file, freeing the buffer.

Load multiple and store multiple. During the operation of most systems, there is a frequent need to move blocks of data at a high rate. This is especially true for the Am29000, because it is necessary to move blocks of data to and from the register file to switch processes and to handle stack cache overflow and underflow. For these reasons, the Am29000 implements load multiple and store multiple operations, which move data at a rate of up to 100M bytes/s, and take advantage of the channel's burst-mode capability, which will be discussed.

Since interrupt latency is an important consideration in real-time environments, and since load multiple and store multiple operations can take a significant amount of time to complete, these operations are allowed to be interrupted. The multiple access is restarted, upon interrupt return, at the point it was interrupted. This capability relies on the same mechanism which restarts the access if an exception occurs.

Instruction fetching and branching. In any RISC, the efficiency of the instruction-fetch mechanism is a major determinant of performance. Instructions must be fetched at the maximum execution rate, and the instruction flow must incur a minimum disruption for successful branches.

The Am29000 achieves its instruction bandwidth by prefetching instructions. A burst-mode protocol simplifies the implementation of a memory system with the required bandwidth. The processor implements delayed branches, and also includes a *branch target cache* to keep the processor pipeline operating efficiently for the majority of successful branches, even in the presence of a relatively high instruction-fetch delay.

Instruction prefetching. The Am29000 normally prefetches an instruction four cycles in advance of execution. An instruction address is transmitted to the instruction memory on every successful branch, and the processor prefetches sequential instructions without the retransmission of an address (although the system can cause the processor to transmit addresses as an option) until the next branch occurs. A four-word *instruction prefetch buffer* handles any temporary mismatches between the instruction-prefetch rate and the instruction-execution rate.

Most of the interference between instruction prefetches and data accesses are eliminated in the Am29000. Instructions are fetched from a separate instruction memory (or a separate port on a single memory), using a dedicated instruction bus.

Contention for the memory management unit (MMU) is eliminated by translating instruction addresses only once—during the execution of branches. During instruction prefetching, the addresses for instruction fetches are computed in the physical address space by a 30-bit instruction fetch pointer. The MMU is thus available for loads and stores. The MMU is needed for instruction prefetches only in the infrequent case that the instruction fetch pointer crosses a virtual page-boundary.

Branching. The Am29000 is capable of executing successful branches in a single cycle. This capability is due to the combination of delayed branches, early target-address computation and branch-condition detection, and the Am29000's branch target cache. Figure 6 illustrates the execution of a successful branch in the Am29000 pipeline, and shows how single-cycle branching is achieved.

All branches in the Am29000 are delayed by one pipeline stage. This means that the instruction following the branch—the delay instruction—is executed regardless of the outcome of the branch. Since the Am29000 provides no other form of branch, the delay instruction must be present; if the delay instruction cannot be useful, it must be a no-op instruction.[12]

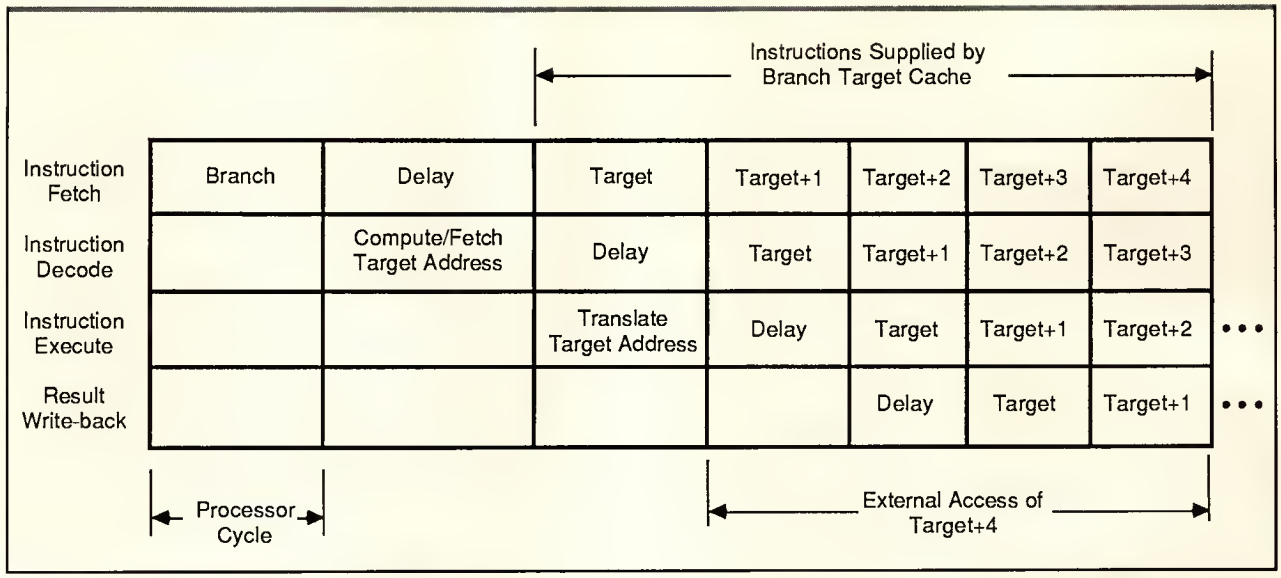


Figure 6. Pipelined execution of successful branches.

Branch conditions in the Am29000 are Boolean data in general-purpose registers, rather than ALU condition codes. This scheme has two benefits. First, it allows the compiler to allocate branch-condition results to registers just as any other instruction result would be. Allocating branch conditions to registers avoids the contention problems associated with a single condition-code register which is affected by many different instructions.[13] Second, since branch conditions are treated as any other instruction result, they are forwarded by the processor pipeline controls as any other instruction result would be. This forwarding process avoids any special paths dedicated to determining the branch outcome in the case where the branch condition is being computed when the branch instruction is in the decode stage.

The Am29000 determines the branch target address during the instruction-decode stage. During the execute stage, the address is used to search for the target instruction in the branch target cache. As Figure 6 shows, the delay instruction is decoded during the execute stage of the branch.

Address translation is performed on the branch target address in parallel with the branch target cache access. Furthermore, the *fetch-ahead adder* (Figure 1) adds 16 to the page offset of the address (that is, the low-order 10 bits). If this add does not cause the address to cross a page boundary, the processor obtains two physical addresses when address translation is complete; the first is the physical address of the branch target instruction, and the second is the physical address of the instruction which is four instructions beyond the target instruction. The latter address is used when the target instruction is found in the branch target cache.

Branch target cache. If the target instruction of a successful branch is in the branch target cache, the first four instructions from the target instruction-stream are supplied by the cache. This helps overcome the delay in the external instruction memory. In essence, instructions from the branch target cache fill branch delay cycles with useful instructions. Since these instructions are at the target of the branch, no compiler scheduling is necessary. Of course, the branch target cache is effective only in those cases where the target of the branch is in the cache. The organization of the branch target cache is shown in Figure 7.

In some respects, the branch target cache is similar to a more traditional instruction cache. The Am29000 could still achieve single-cycle branches if the branch target cache were replaced by an instruction cache. However, the branch target cache avoids the fragmentation problems of a traditional cache (since the cached block is fixed at four instructions), and avoids wasting on-chip memory by storing instructions which might as well be fetched from an external memory.

When a branch target is found in the branch target cache, it is necessary to begin fetching instructions externally at the location which is four instructions beyond this target. This is the reason for the fetch-ahead adder described above. By adding 16 to the branch target address, the fetch-ahead adder allows the Am29000 to immediately begin prefetching instructions for the branch target instruction stream.

Interrupts and traps. As with most other features of the Am29000, the interrupt mechanism was defined to allow a high degree of flexibility to users. By keeping the amount of predefined interrupt processing to a minimum, the Am29000 also holds the fixed costs of interrupt handling to a minimum.

Interrupt mechanism. The basic interrupt mechanism of the Am29000 is such that only one register—the *current processor status register*—is automatically saved when an interrupt or trap is taken. This register is saved in another processor register, called the *old*

processor status register. Other registers containing the state of a process are "frozen." When a register is frozen, it is not changed unless it is explicitly written by an instruction. This allows an interrupt/trap handler to execute without disturbing the state of the interrupted process.

For certain types of interrupts and traps, the interrupt/trap handler can execute without taking the time to save or restore any process state. The state is simply left in processor registers during the execution of the handler. This approach is most appropriate for short system service routines which are frequently executed, such as the routines which load *translation lookaside buffer* (TLB) entries from system page tables. These routines can be executed without the overhead of saving and restoring state.

To provide for nested interrupts and traps, the interrupt/trap handler must save the state of the interrupted process, and reenable interrupts and traps. Since this is accomplished by normal instruction sequences, the amount of state saved, and the data structures used to save and restore state, are system-dependent. For example, the state can be saved in and restored from general-purpose registers (global registers are adequate for saving the state of up to eight processes). Alternatively, the state can be saved in an interrupt stack, or other traditional interrupt data structure, in memory.

Interrupt latency and response. The Am29000 is designed to allow the interrupt latency and response times to be largely under the control of the system designer. The fact that all but four Am29000 instructions execute in a single cycle solves most of the problems. Two of the multiple-cycle instructions perform interrupt-return sequences, which cannot be interrupted. The remaining two multiple-cycle instructions —Load Multiple and Store Multiple—are interruptible, so that they do not have much effect on interrupt latency.

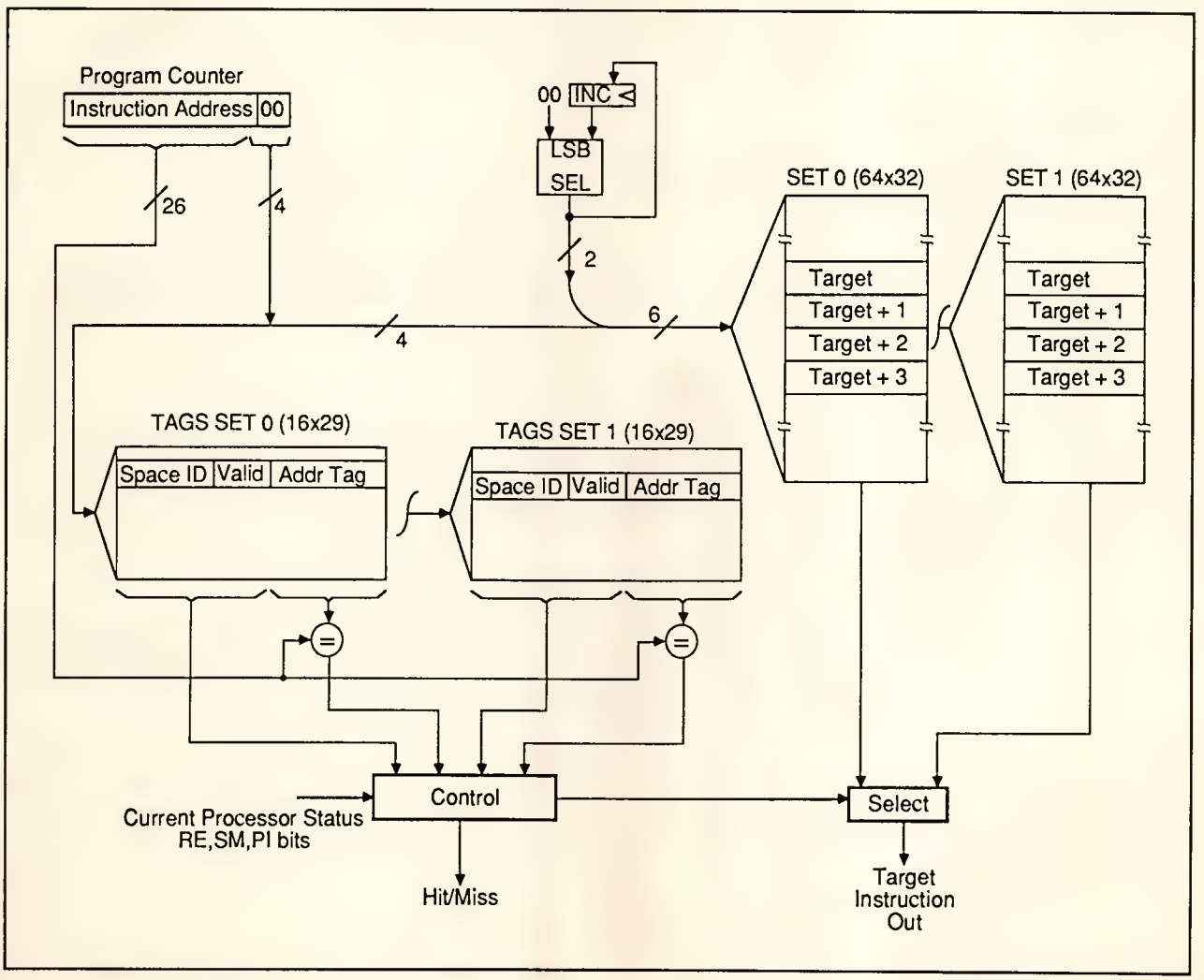


Figure 7. Branch target cache.

Memory management. The MMU in the Am29000 consists mainly of a 64-entry TLB, which translates a 32-bit virtual address into a 32-bit physical address in a single cycle. The TLB performs address translation (see Figure 8) during the instruction-execution stage of the loads, stores, and branches. The resulting physical address is available externally on the next cycle.

The Am29000 MMU is not dedicated to any particular memory-management architecture. If the MMU cannot perform an attempted address translation, it simply causes a TLB miss trap. The trap handler must search the system page table(s), and either place the proper entry into the TLB or signal a page fault.

The decision to use software TLB reload on the Am29000 was based on several considerations. Primary among these was the difficulty in choosing a suitable architecture for the diversity of intended applications for the Am29000. These applications range from embedded controllers, using only primitive memory-management functions, to large, multiuser systems with much more sophisticated memory management. Using software TLB reload, any data structures and algorithms can be used in the memory-management software; these can be chosen as suitable for the particular application.

Software TLB reload also has a lower hardware cost than hardware TLB reload. Regardless of the memory-management architecture, a processor is idle during TLB reload, since there is nothing else for it to do. In a sense, it is wasteful to define logic, sequencing, and memory-access mechanisms for TLB reload which only duplicate the function of a general-purpose resource (such as the processor) which is sitting idle. This is especially true for the Am29000, which is capable of single-cycle instruction execution, overlapped page-table accesses, and very low trap overhead. Software TLB reload on the Am29000 bears a strong resemblance to microcoded TLB reload on other processors.

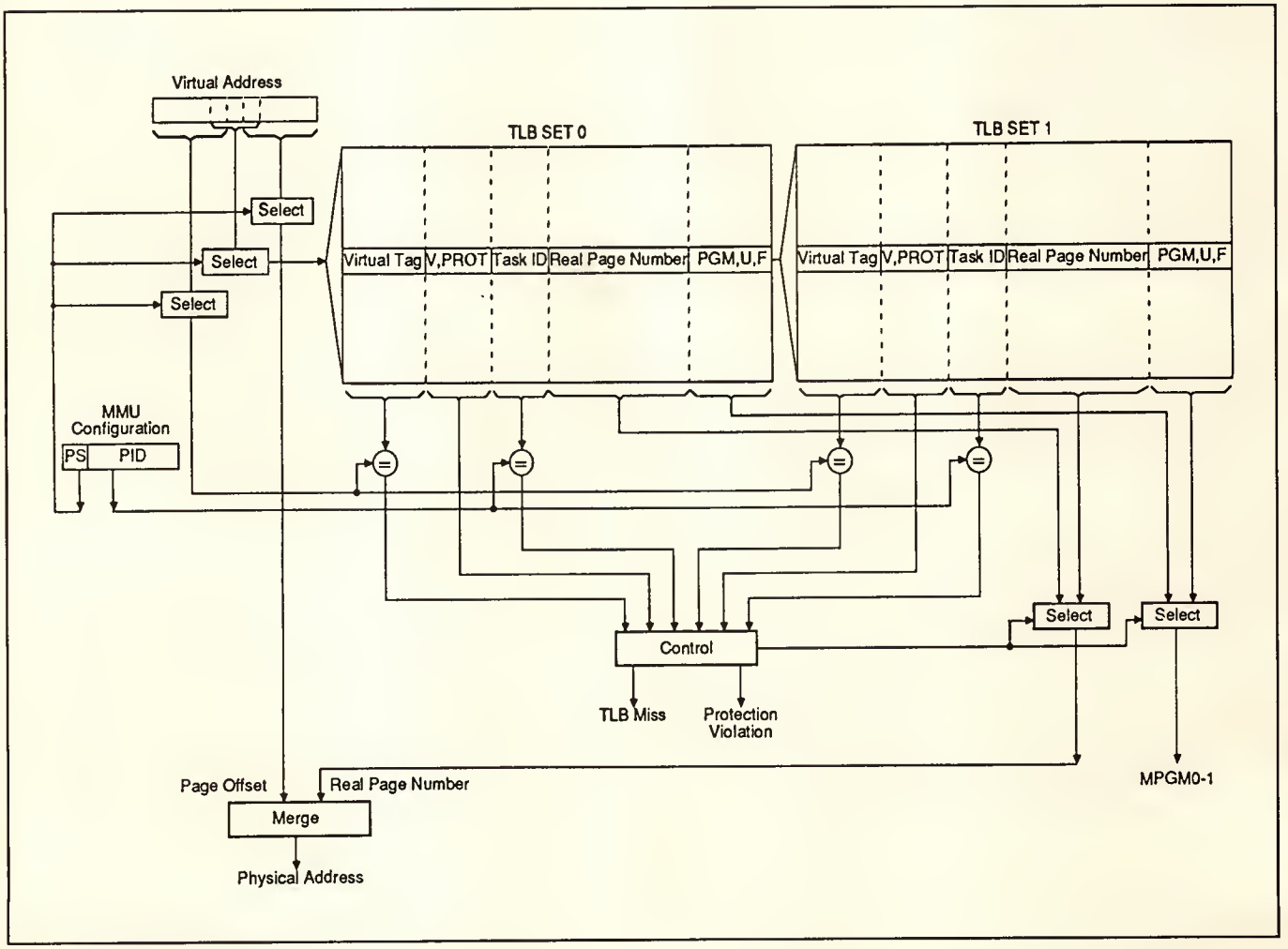


Figure 8. Address-translation process.

Channel. The Am29000 performs all external accesses over a three-bus channel. This channel includes an address bus, an instruction bus, and a data bus. Figure 9 shows the connection of these buses in a typical system. The three-bus architecture of the Am29000 resulted from a desire to provide the maximum data and instruction bandwidth without multiplexed buses, and with a reasonable pin count. The address bus is shared for instruction and data accesses in a manner which does not introduce contention between these accesses, as further explained.

Access protocols. The Am29000 channel implements three access protocols, which apply (independently) to both instruction and data accesses. For each type of access—simple, pipelined, and burst-mode—the protocol is defined to allow both the lowest possible delay and the highest possible bandwidth. In particular, the protocol signals are not pipelined; the state of the control signals on a given cycle indicate the state of the channel for that cycle only.

For simple accesses, the processor holds the address for the access valid throughout the entire access cycle. The access can complete either in the first cycle that the address appears on the address bus, or on a later cycle. In any event, the address is held until the slave indicates that the access is complete. Simple accesses are used for high-speed devices and memories, and for simple devices (such as initialization ROMs) which have relatively long access times but which are accessed infrequently.

The address for a pipelined access is transmitted while a previous access is still in progress. This allows address transmission and decoding to be overlapped with other portions of the access. Pipelined accesses reduce the utilization of the address bus, and can yield a higher throughput than simple accesses.

For burst-mode accesses, multiple instructions or data words are accessed with a single address transmission. A starting address is transmitted once at the beginning of the access, and all other addresses are obtained by incrementing this address. Burst-mode accesses are normally used for instruction prefetching, Load Multiple, and Store Multiple. The rate of flow of instructions or data—and the number of accesses performed—are controlled dynamically by the burst-mode protocol (however, the processor does not exercise any control on the flow of data, since this is to or from processor registers, which are always available and valid).

Burst-mode accesses allow instructions and data to be transferred at a rate of 100M bytes/s at 25 MHz. This bandwidth is easier to achieve than it is for simple and pipelined accesses, since there is no address transfer or decode. Burst-mode accesses also allow the memory design to take advantage of high-bandwidth features, such as static-column accesses. Burst-mode I/O accesses are also possible, if advantageous for a particular system.

In a high-performance system design using the Am29000, instructions are fetched, using burst-mode accesses, over the instruction bus. The address bus and data bus are free for random data accesses during program execution. The address bus is used for instruction addresses only when branches are taken. Since loads, stores, and branches are caused by separate instructions, the fact that these instructions are issued on separate cycles naturally schedules the use of the address bus between instruction and data accesses. Using this approach, there is no contention between instruction and data accesses, and no system component need operate at a cycle time greater than the processor cycle.

Exception and error reporting. Any instruction or data access can be ended with an error indication from the system. An instruction access which ends in

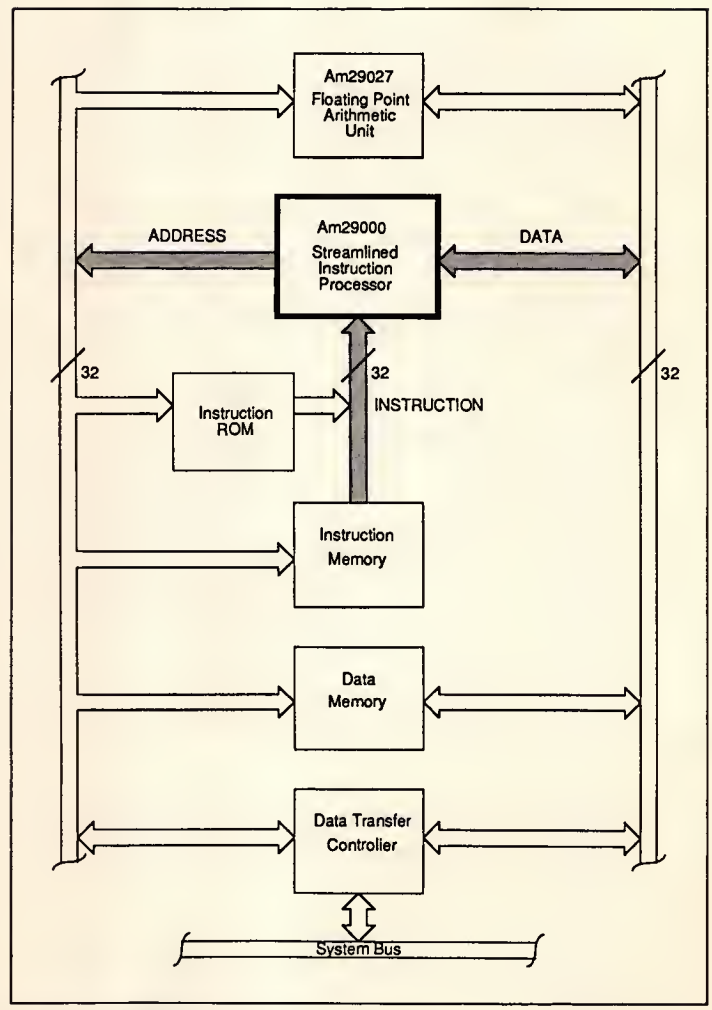


Figure 9. Three-bus channel.

Table 2.
CPU status and control signals.

STAT2	STAT1	STAT0	Condition
0	0	0	Halt or step modes
0	0	1	Pipeline hold mode
0	1	0	Load test instruction mode
0	1	1	Wait mode
1	0	0	Interrupt return
1	0	1	Taking interrupt or trap
1	1	0	Nonsequential instruction fetch
1	1	1	Executing mode

CNTL1	CNTL0	Mode
0	0	Load test instruction
1	0	Step
0	1	Halt
1	1	Normal

error causes an *instruction access exception trap* if the processor attempts to execute the corresponding instruction (that is, if it does not branch before executing the instruction). A data access which ends in error always causes a *data access exception trap.*

Both the instruction access exception and data access exception traps are unmaskable, so that they may be used to report hardware errors regardless of the processor's operational mode. However, the access creating the trap is restartable in both cases, employing the same mechanisms used for restarting an access after a normal exception (such as a TLB miss). Thus, these errors can also be used to signal recoverable system exceptions.

Testing and reliability. The Am29000 contains provisions for testing the processor chip and related system components, and for guaranteeing correct system operation to the degree desired for the system application. The processor structure made designing for testability very simple, requiring only a small amount of additional logic for a high degree of testability. The test/development interface of the Am29000 also includes provisions for hardware debug in the target-system environment (without an in-circuit emulator), and for testing other system components.

CPU status and control lines. Five pins on the Am29000 give information about processor operation and allow complete external control over processor instruction execution. These pins, and their functions, are shown in Table 2.

The CPU status outputs (STAT2-STAT0) indicate the operational mode of the processor's instruction-execution stage on the previous cycle. All indicated modes with the STAT2 output High correspond to active processor operation. Those modes with the STAT2 output Low correspond to idle or halted modes. This allows external measurement of processor performance, and provides a means for an external watchdog timer to monitor processor idle states. An error can be reported if the processor is idle for more than some maximum expected amount of time.

The CPU control inputs (CNTL1-CNTL0) allow external control over processor operation. By using these controls, an external tester or hardware-development system can halt the Am29000, step through pipeline operation, and supply instructions for execution. Each of these operations is defined to require no support in any other system component, and is defined so that normal operation can resume from the point at which the tester or hardware-development system intervened.

Hardware debug. Using the CNTL1-CNTL0 inputs, in conjunction with the instruction bus, the Am29000 can be made to execute any instruction, regardless of its operational mode or the state of its instruction fetch and execution units. The *test/development interface* provides a capability similar to microinstruction jamming on microprogrammed processors. The Am29000 interposes no logic between the instruction bus and the instruction register (none is needed because of the regularity of the instruction set). Consequently, it is trivial to allow an instruction to be directly loaded into this register for execution.

Since any instruction can be executed via the test/development interface, processor state can be examined and modified via this interface. For example, load and store instructions can be used to modify and inspect the contents of general-purpose registers, with the relevant data appearing on the data bus; these instructions can be stepped, so the hardware-development system need not keep up with the processor's data rate. The contents of general-purpose registers can be moved to and from other special-purpose registers and the TLB, so these can be inspected and modified as well. It is also possible for the Am29000 to access system memories and devices on behalf of the hardware-development system.

Once all desired instructions have been executed via the test/development interface, the Am29000 can resume its original sequence of operations. This provides a general capability to halt the processor, inspect and modify data, and continue—operations which occur routinely during system debug. In addition, all of this is possible without requiring any special modifications to the system under test.

Hardware testing and master/slave checking. For testing in a manufacturing environment, the Am29000 provides a test mode in which processor outputs are in a high-impedance state. This allows

processor outputs to be driven directly by a tester for the purposes of testing system hardware.

The test mode is also used in the implementation of master/slave checking. All Am29000 outputs have a comparator which compares the output of an enabled off-chip driver with the signal being supplied internally to the driver. If the output of the driver does not agree with the input, an error is signalled externally. No other action is taken; the external error signal can have any desired effect on the processor or system.

In a master/slave configuration, two Am29000's are connected in parallel, with outputs of the slave processor disabled by the test mode. In this configuration, the slave processor checks the outputs of the master processor against its own (disabled) outputs. The slave processor thus checks that both it and the master processor have consistent operation, providing a good degree of local reliability.

The Am29000 is, in many respects, a second-generation RISC. We have consistently found that the desired goals of performance and flexibility could be met with a simple, straightforward set of architectural features. These results have confirmed and extended those achieved for previous RISCs.

The performance of the Am29000 has significantly exceeded our initial expectations. By concentrating on the efficiency of the processor pipeline during design, we were able to take maximum advantage of a single-cycle instruction-execution rate.

References

1. J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-To-End Arguments in System Design," *ACM Trans. Computer Systems*, Nov. 1984, pp. 277-288.
2. D.A. Patterson and D.R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News*, Oct. 15, 1980, pp. 25-33.
3. J.L. Hennessy et al., "Hardware/Software Tradeoffs for Increased Performance," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, Mar. 1982. Published as *SIGArch News*, Mar. 1982, pp. 2-11 and as *SIGPlan Notices*, Apr. 1982, pp. 2-11.
4. G. Radin, "The 801 Minicomputer," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, Mar. 1982. Published as *SIGArch News*, Mar. 1982, pp. 39-47, and as *SIGPlan Notices*, Apr. 1982, pp. 39-47.
5. D.W. Wall, "Global Register Allocation at Link Time," *Proc. SIGPlan 86 Symp. Compiler Construction*, June 1986. Published as *SIGPlan Notices*, July 1986, pp. 264-275.
6. G.J. Chaitin et al., "Register Allocation via Coloring," *Computer Languages*, Vol. 6, No. 1, 1981, pp. 47-57.
7. D.A. Patterson and Carlo H. Sequin, "A VLSI RISC," *Computer*, Sept. 1982, pp. 8-21.
8. D.R. Ditzel and H.R. McLellan, "Register Allocation for Free: The C Machine Stack Cache," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* Mar. 1982. Published as *SIGArch News*, Mar. 1982, pp. 48-56 and as *SIGPlan Notices*, Apr. 1982, pp. 48-56.
9. J.C. O'Quin, "The IBM RT PC Subroutine Linkage Convention," *IBM RT Personal Computer Technology*, IBM Corporation, Austin, Tex., 1986, pp. 131-133.
10. P. Chow, *MIPS-X Instruction Set and Programmer's Manual,* Rev. 1.5, Computer Systems Laboratory, Stanford University, Stanford, Calif., Sept. 24, 1985.
11. J.L. Hennessy and T.R. Gross, "Postpass Code Optimization of Pipeline Constraints," *ACM Trans. Programming Languages and Systems*, July 1983, pp. 422-448.
12. T.R. Gross and J.L. Hennessy, "Optimizing Delayed Branches," *Proc. Micro-15*, IEEE, Piscataway, N.J., Oct. 1982, pp. 114-120.
13. M. Auslander and M.E. Hopkins, "An Overview of the PL.8 Compiler," *Proc. SIGPlan 82 Symp. Compiler Construction*, June 1982. Published as *SIGPlan Notices*, pp. 22-31.

Mike Johnson is section manager for microprocessor strategic development at Advanced Micro Devices. He heads the group responsible for the architecture definition and specification of the Am29000. Other responsibilities of the group include support of chip design, marketing, and tool development. Before joining AMD, he worked for eight years at IBM in Austin, Texas, where he was a major contributor to the definition and design of the ROMP microprocessor and instrumental in the development of the IBM RT Personal Computer.

Johnson received a BSEE in 1975 and an MSEE in 1976—both from Arizona State University. He has recently been pursuing post-MS studies in electrical engineering at Stanford University.

Questions about this article can be directed to Johnson at Advanced Micro Devices, 901 Thompson Pl., PO Box 3453, M/S 70, Sunnyvale, CA 94088.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 153 Medium 154 Low 155

THE 80387 AND ITS APPLICATIONS

David Perlmutter and Alan Kin-Wah Yuen
Intel Corporation

While compatible with the 80287, the 80387 works four to six times faster, making it a good bet for use in engineering workstations and real-time, embedded systems.

In today's microprocessor systems, floating-point coprocessors are becoming the standard option. This trend began with a need for more floating-point processing power, the industry standardization that came about with the introduction of the 8087 floating-point coprocessor for the 8086, and the subsequent support by popular software. Intel Corporation contributed further to this trend when it introduced the 80287 coprocessor as an option for the 80286 microprocessor.

The 80387 coprocessor, designed to complement the 80386 32-bit microprocessor, represents the highest floating-point performance in the 8087 family. It offers four to six times the performance of the 80287. Implemented in the CHMOS III double-metal process, the same fabrication process for the 80386, the 80387 achieves 16- and 20-MHz performance levels. The combined microsystem provides the most powerful, closely coupled microprocessor configuration with the largest software base available on the market.

Since the advent of the 8087, two major classes of on-chip floating-point solutions have emerged. One class borrows the RISC concept from microprocessor design, performing very fast basic operations (Add/Subtract, Multiply, Divide, Compare) in single or double precision. This class requires software to support exception handling and data format conversions (integer or BCD to floating-point formats and vice versa). Complex trigonometric, logarithmic, and exponential operations are not supported in hardware. All these operations entail a significant amount of software development, maintenance cost, and additional product development time.

The second design approach presents a more comprehensive solution on chip. These floating-point coprocessors provide more complex instruction extension for the CPU and automatically take care of exceptional cases, rounding, and precision control. Such solutions usually keep intermediate results in 80-bit, extended-precision format, thus reducing the chances of overflow and underflow and preserving precision. The extended format gives better accuracy. Because their microcode is written to take the best advantage of the hardware, these coprocessors perform floating-point instructions much faster than software emulators. Therefore, in addition to the reduction of complexity and cost of software development, speed increases significantly. Intel numeric processor extensions pioneered this class of floating-point solutions.

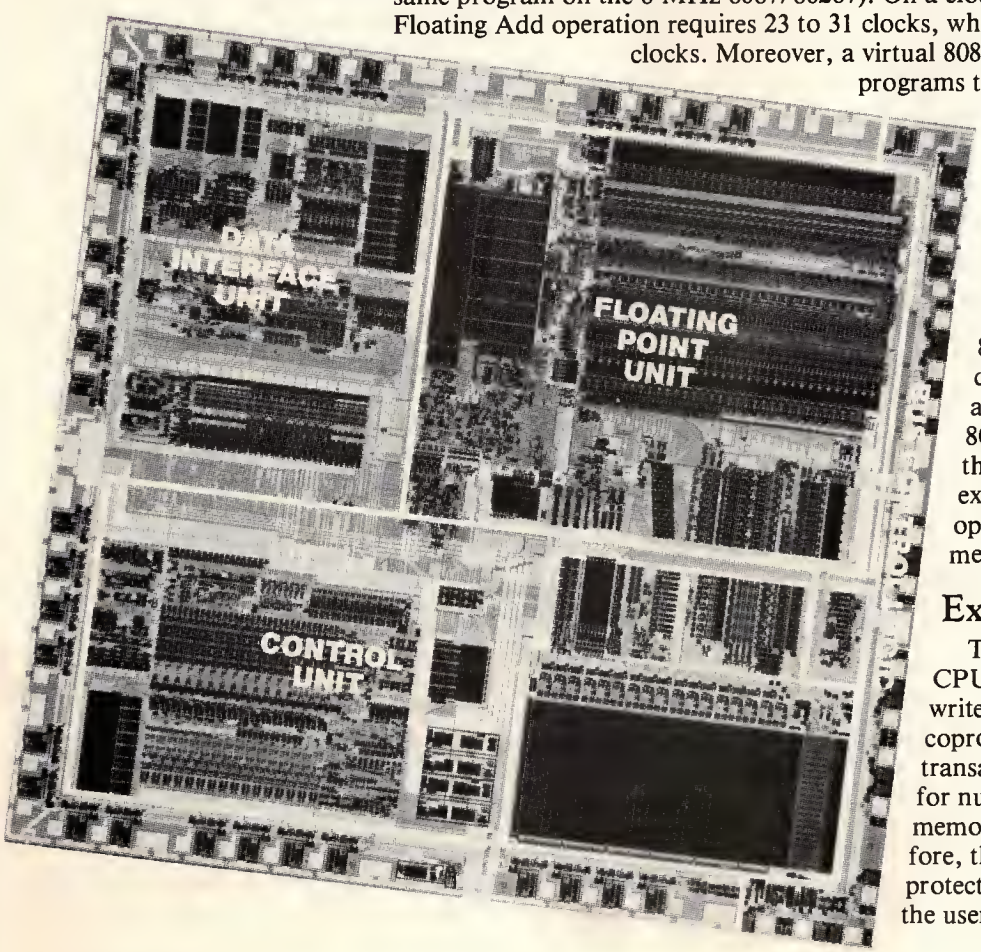
0272-1732/87/0800-0042$01.00 © 1987 IEEE

IEEE standard implemented. The 80387 fully supports the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic,* which defines the basic algorithms and implementation-independent rules for performing floating-point operations. This standard supports software portability and interfacing between different machines. It also defines the data types to be supported, their accuracy, rounding control, and exception-handling support.

The 80387 fully supports all basic floating-point operations and all precision and rounding modes required by the standard. It also handles internally all exceptions defined by the standard, thus freeing external software from filtering and handling the exceptional cases. A recommendation of the standard to round constants is also fulfilled.

In addition to basic operation support, the 80387 converts data types from integer and BCD to floating point and vice versa and variables from one precision to another. It calculates remainders, SQRT, and scaling of operands and supports basic trigonometric, logarithmic, and exponential (base 2) operations. Such a variety of instructions gives the programmer a lot of flexibility.

Family compatibility. The 80387 executes object codes compiled for its predecessors. This means that any compiled program written for 8087/80287 can be run without the need for recompilation. All operating modes of the 80386 are supported. Both 16-bit and 32-bit programs can take advantage of the high performance of the 80387. For example, the 80387 (operated at 16 MHz) executes the Whetstone benchmarks in double precision at 1.5 MWhetstones/s (versus 0.3 MWhetstones/s for the same program on the 8-MHz 8087/80287). On a clock count basis, the 80387 Floating Add operation requires 23 to 31 clocks, while the 80287 requires 70 to 100 clocks. Moreover, a virtual 8086 mode of the 80386 allows DOS programs to use the full performance of the 80387 in a 32-bit operating system environment.



The 80387 supports a super set of the 8087/80287 instructions with a variety of enhancements such as increased operand range for transcendental functions. It connects directly to the 80386 through an optimized and dedicated coprocessor interface in a similar fashion to the way the 8087 and 80287 are connected to their host processors. The 32-bit external interface allows faster operand transfers and larger memory for access.

External architecture

The 80387 interfaces to the 80386 CPU as a slave coprocessor. The CPU writes opcodes and operands to the coprocessor and performs memory transactions for it. All memory accesses for numeric operands use the full 80386 memory management hardware. Therefore, the integrity of the 80386 memory protection mechanism is preserved, and the user's memory management scheme

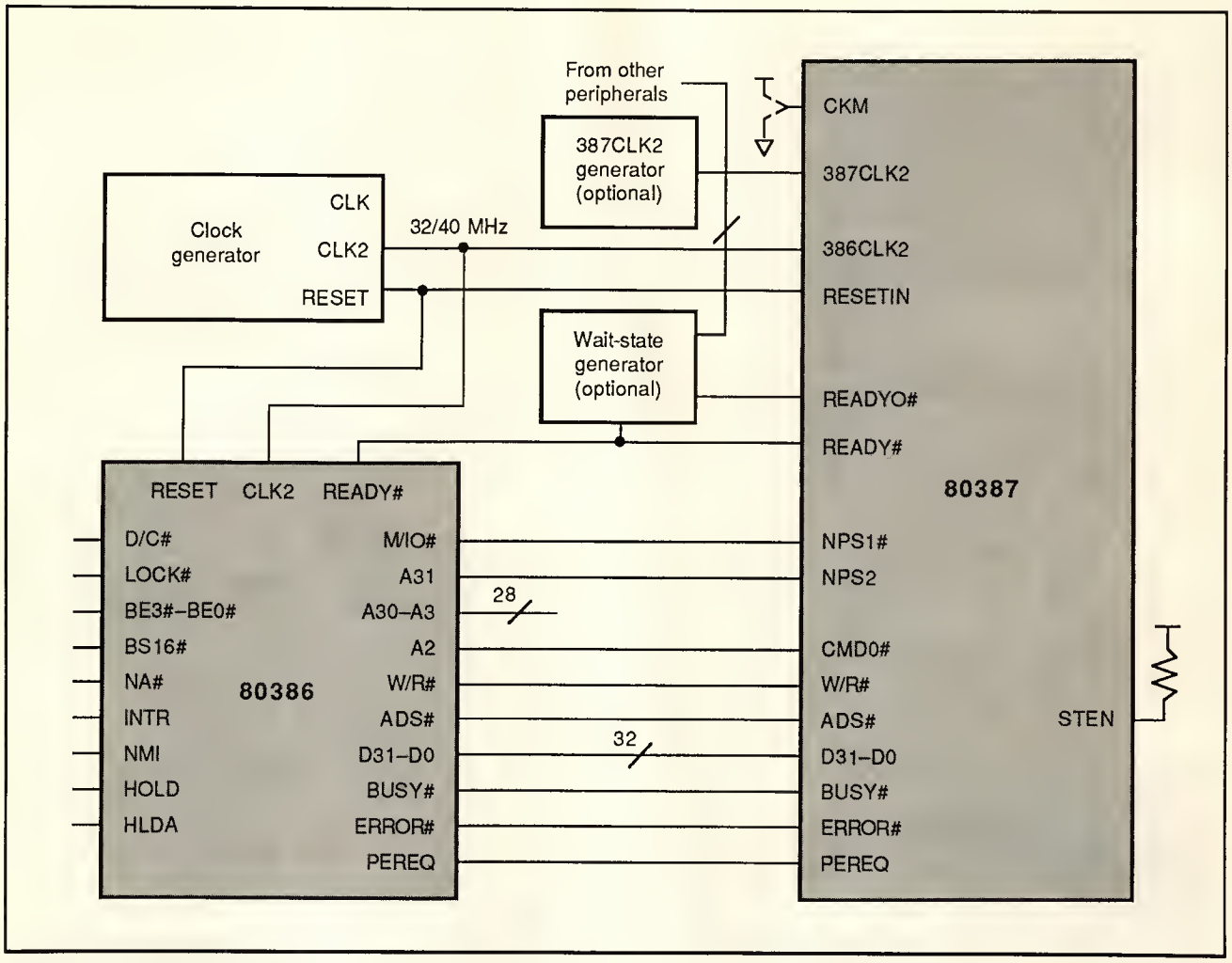


Figure 1. 80386/80387 system configuration.

is simplified. Figure 1 illustrates this interface; Table 1 describes the pins and their functions.

The address and data buses are 32 bits wide to match the high bus bandwidth of the 80386. The following signals select the coprocessor:

- NPS1# (connected to M/IO# of 80386) and
- NPS2 (to A31 of 80386) with STEN pulled high.

CMD0# (connected to A2 of 80386) distinguishes between opcode and data cycles initiated to the 80387. Other system control signals include W/R# (write/read), ADS# (address/data strobe), and READY#. READYO# is the active-low ready output from the 80387. BUSY#, ERROR#, and PEREQ denote when the 80387 is busy executing the current instruction, when an error occurs, and when the 80387 is ready for operand transfer for the 80386, respectively. A key signal, the CKM, indicates to the 80387 that the synchronous mode of operation is desired. In such a case, the internal execution unit uses the same clock as does its bus interface and the 80386. Otherwise, a separate clock is required to connect to the 387CLK2 input.

System interface. In defining the system architecture, we concentrated on optimizing two major factors: overall system performance and system cost.

The 80387 resides on the 80386 local bus to serve 80386 requests. It directly decodes the control signals and traces all bus cycles on the bus. Bus cycle tracing is required to recognize pipelined data cycles to the 80387. ADS#, READY#, and control signals must be monitored to recognize such cycles. A synchronous interface is required to correctly sample the ADS# and READY#. No glue logic is required. This not only reduces system cost but also increases system reliability. The simplicity of the interface eases the system design.

The 80387 can be envisioned as two different devices in one package:

Table 1.
80387 I/O pin description.

Pin	Function
386CLK2	80386 CLocK 2 clocks the bus interface of the 80387, which is always synchronous to the 386 local bus. If the 80387 runs in synchronous mode, this clock is also used to clock the execution and the rest of the circuits. It uses the same system clock as the 80386. For a 16-MHz system, it operates at 32-MHz frequency.
387CLK2	80387 CLocK 2 clocks the execution unit of the 80387 if running in asynchronous mode. It has to use a separate clock in this mode. At the execution speed of 16 MHz, it operates at 32-MHz frequency.
CKM	ClocKing Mode is a strapping option. When tied high, the 80387 is intended to run in synchronous mode; when tied low, in asynchronous mode.
RESETIN	System reset
PEREQ	Processor Extension Request indicates whether the 80387 is ready for data transfer or not.
BUSY#	Indicates that the 80387 is busy executing an instruction.
ERROR#	Indicates that a math error has occurred.
READYO#	Ready Output from the 80387 for opcode and data transfers between the 80386 and 80387
READY#	Bus Ready input
STEN	STatus Enable; when active, all 80387 outputs are enabled.
NPS1#	Numeric Processor Select #1
NPS2	Numeric Processor Select #2
CMD0#	CoMmanD distinguishes between opcode and data transfers.
ADS#	ADdress Strobe; the 80387 uses this 80386 output to monitor bus activities. It indicates the beginning of any new bus cycle.
W/R#	Write/Read
D31-D0	A 32-bit data bus is connected directly to the 80386 data pins.

- the Bus Control Logic unit, which serves as a dedicated bus controller for the 80387, and
- the 80387 core, which performs the computations.

A block diagram of the 80387 is shown in Figure 2 on the next page.

The use of a 10-byte-deep FIFO between Bus Control Logic and the core permits the 80386 to process Reads/Writes with the minimal wait states possible (0 in Writes, 1 in Reads). It also facilitates the pseudo-synchronous mode of operation in which the execution unit runs asynchronously to the bus interface. However, the 80387 bus interface always runs synchronously with the 80386. This mode of operation is effected by the input signal CKM.

Interface protocols

The 80387 instructions are an integral part of the 80386 instruction set. When the 80386 recognizes an 80387 instruction, it partially decodes it and initiates bus transactions for and with the 80387. The 80386 differentiates among the following instruction categories (only major categories are listed):

1) *Instructions that load an operand from the memory with the operand length in the range of two to 10 bytes.* These include all the load instructions as well as all arithmetic operations that need one operand from memory. The destination for a load operation and the second operand for a two-operand arithmetic operation is assumed to be at the top of the floating-point stack. Examples are FILD MEMORY[SI], FADD [BP].UPPERLIMIT, and FBLD BCD_NUMBERS. This category is referred to as "load type" instructions.

2) *Instructions that store an operand in the memory with the operand length in the range of two to 10 bytes.* These store instructions also convert internal, 80-bit extended format to the precision format as implied by the variable type. Examples of this category are FIST MEMORY[SI] and FBSTP BCD_NUMBERS. We call this category "store type" instructions.

3) *Numeric instructions that involve only the stack elements.* ST, the top of the stack, is always the im-

plied stack element when omitted from an instruction. Examples of this category are FPATAN, FSTP ST(2), and FMULP ST(5),ST. We call this category "nonoperand transfer type" instructions.

4) *Administrative instructions that support the operating systems and exception handlers.* These instructions do not involve any operand transfer from any source. Examples are FNINIT (FINIT without WAIT prefix) and FNCLEX (FCLEX without WAIT prefix). This category is referred to as "administrative type" instructions.

5) *Other categories exist that involve saving and restoring the 80387 state or its partial state.* These instructions are basically similar to the load type and store type instructions. They differ in their longer operand length. Examples of these instructions are FLDENV [BP] and FNSAVE [BP] (FSAVE without WAIT prefix). This category is referred to as "state save/restore type" instructions.

(See Table 2 for a list of the instructions discussed here.)

Synchronization becomes necessary when both the CPU and the coprocessor execute instructions concurrently. This state includes instruction and data synchronization and error handling. The 80386/80387 synchronization is implemented in three hierarchical levels:

- *Instruction synchronization.* Since the duration of the 80387 instructions is relatively long, the 80386 could try to write the next 80387 instruction before the previous one is completed. The 80386 and 80387 handle synchronization automatically.
- *Data synchronization.* Within the execution of an 80387 instruction, data transfer between the 80386 and the 80387 must be synchronized. The system also handles these transfers automatically.
- *Error synchronization.* When an exception is unmasked and an error condition is detected by the 80387, the corresponding exception bit and error summary bit in the 80387 status word are set. This step causes the ERROR# output to become active

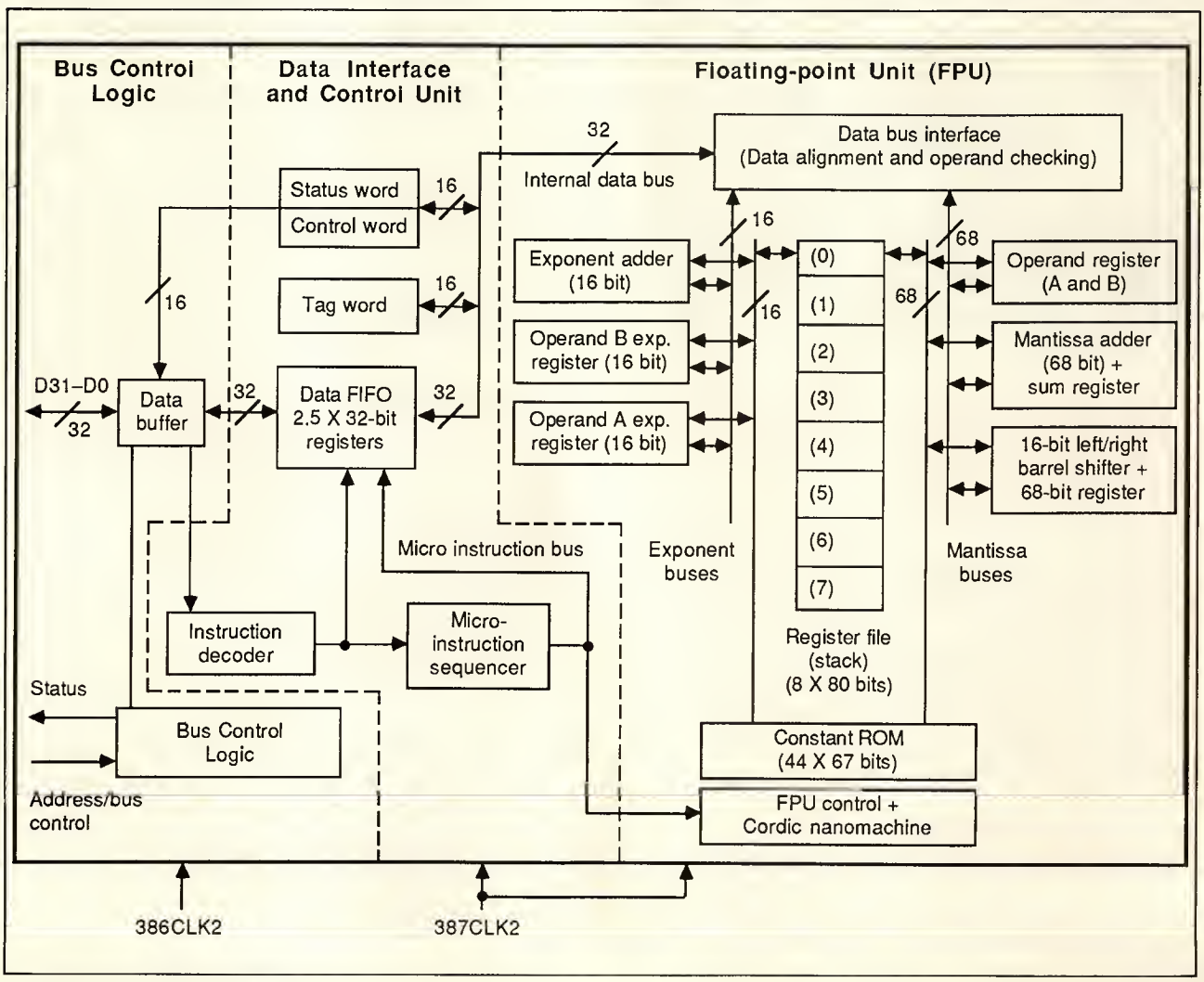


Figure 2. 80387 block diagram.

and hence gets the attention of the CPU. Since the error pointers to the failing floating-point instruction and operand are saved in the 80386, application software can locate the error point.

The FWAIT instruction and three signals, BUSY#, PEREQ, and ERROR#, are all that would be required to preserve the various types of synchronization.

Instruction synchronization. BUSY# is used to synchronize the parallel instruction execution between the 80386 and the 80387. The 80387 signals that it is busy by activating the BUSY# signal and that it is ready to execute the next instruction by deactivating the BUSY# signal. By testing its BUSY# input, the 80386 can decide whether it can write the next 80387 instruction or wait until the 80387 is ready. While waiting, the 80386 honors interrupt requests.

In the case of load type instructions (excluding transfers of two or 10 bytes), the 80386 writes the next (floating-point) opcode in line before the 80387 completes the current instruction. It waits until BUSY# has been deactivated before transferring the operand for the new instruction. On the other hand, once the 80387 frees itself from the current instruction, it starts executing the new instruction. The execution is prohibited from proceeding any further only if it has to wait for a memory operand. The 80387 will activate its BUSY# output only when the last operand word has been written to it (in contrast to the other cases where BUSY# is activated when the instruction is written). The execution of the new instruction can be aborted by the 80386 (by writing a new opcode instead of an operand) in case any exception has occurred in executing the current instruction.

Administrative type instructions do not use this protocol, and the 80386 does not check its BUSY# input before writing the opcode to the 80387. The 80387 processes these instructions in parallel with any other instruction already being processed by the core at that time.

Table 2.
Partial list of 80387 instructions (used in the text).

Instruction	Function
FLD	Loads real numbers from memory/stack to stack top (ST) or ST(0); ST(1) is the next to the top of the stack.
FILD	Loads integer numbers from memory/stack to ST
FBLD	Loads BCD numbers from memory/stack to ST
FST	Stores ST to memory/stack in real format
FSTP	Similar to FST—also pops ST off the stack
FBSTP	Similar to FSTP—also stores in BCD format
FIST	Stores ST to memory/stack in integer format
FADD	Adds ST and memory/stack operand. For stack-stack operations, the first operand after mnemonics is the destination.
FMUL	Multiplies ST with memory/stack operand; similar comment for stack-stack operations
FMULP	Similar to FMUL—also pops ST
FSIN	Calculates the sine of ST, including operand reduction
FCOS	Calculates the cosine of ST, including operand reduction
FSINCOS	Generates sine and cosine in one instruction
FPATAN	Partial arc tangent of ST(1)/ST
FSTENV	Saves the internal states of the 80387 in memory
FLDENV	Restores the internal states from memory
FSAVE	Saves the internal states and stack contents in memory
FRSTOR	Restores the internal states and stack contents from memory
FINIT	Initializes the 80387
FCLEX	Clears the exception flags in the status word
FWAIT	Ensures the completion of the current floating-point instruction before executing any other instruction; also makes the CPU check the status of its ERROR# input.

Most arithmetic operations use two operands, both of which could be stack elements; however, at least one of them must be the top of stack ST. The implied destination is always the ST, unless otherwise indicated. The complete list of 80387 instructions is documented in the *80387 Datasheet* and the *80387 Programmer's Reference Manual.*

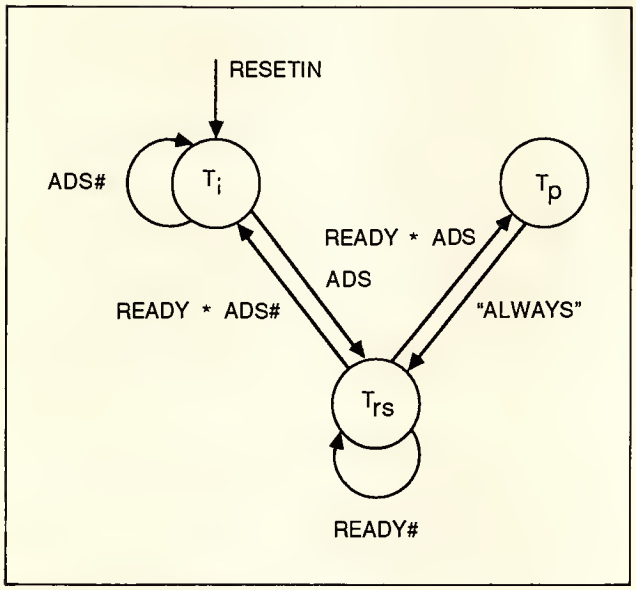


Figure 3. Bus state diagram.

Data synchronization. PEREQ is used to synchronize the data transfers. By testing the state of PEREQ, the 80386 can tell whether the 80387 is ready for operand transfer or not. As soon as the 80387 is ready for operand transfer, it activates PEREQ. It deactivates the signal only when its FIFO becomes full (in the case of load type instructions) or empty (in the case of store type instructions). A good example is a store type instruction in which the 80387 performs data type conversion for the data on the stack before storing it and hence is not ready to transfer. In this case PEREQ is deactivated.

Another situation in which the use of PEREQ is essential is the 80387 execution of state save/restore type instructions. The transferable data is much longer than the depth of the data FIFO can accommodate. In this situation PEREQ regulates dataflow into and out of the FIFO. In load type instructions (excluding loads of two or 10 bytes), the 80387 FIFO is always ready to accept data, and the use of PEREQ is not needed.

Error synchronization. On some occasions users would like to handle a numeric exception in a different way than the default handling of the 80387. To support such applications, the 80387 drives its ERROR# output whenever it encounters an unmasked exception (that is, an exception users want to handle differently). The 80386 samples this signal whenever it encounters an exception-checking 80387 instruction, for example, FWAIT. If activated, the 80386 is interrupted. This ERROR# signal can be deactivated only by a user application using FNCLEX or FNINIT instructions.

Most store instructions must be followed by an FWAIT so that the ERROR# signal will be sampled by the CPU in case of an error in storing the result. Otherwise, the next instruction might use invalid data. An example of the use of an FWAIT instruction is:

```
FST mem        ; store single-precision floating-point
                 result
FWAIT          ; check 80387 ERROR#
MOV eax, mem; before using the result
```

If no FWAIT is inserted after FST, the MOV instruction might end up using the invalid data stored in mem if there were indeed a math error.

Bus cycles. We describe the bus operation in terms of an abstract state machine shown in Figure 3. Bus cycles are more complex because of the 80386 pipelined bus. The pipelined bus allows more time to respond to new bus cycle addresses. Nonpipelined cycles are bus cycles that get started after the previous bus cycle has been terminated by READY#. Pipelined cycles get started before the end of the previous bus cycle.

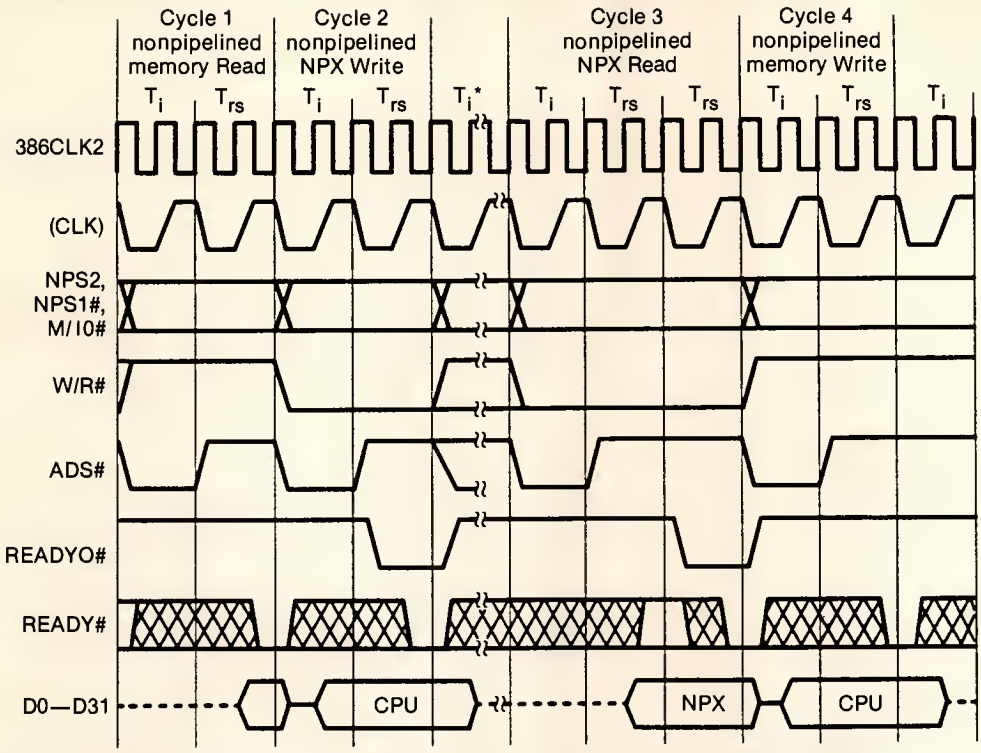
- T_i is the idle state. This is the state of the bus logic after RESET, the state to which Bus Control Logic returns after every nonpipelined cycle, and the state to which bus logic returns after a series of pipelined cycles. New cycles start with new addresses.
- T_{rs} is the READY# sensitive state. Different types of bus cycles may require a minimum of one or two successive T_{rs} states. The bus logic remains in the T_{rs} state until READY# is sensed, at which point the bus cycle terminates. Any number of wait states can be implemented by delaying READY#, thereby causing additional successive T_{rs} states.
- T_p is the first state for every pipelined bus cycle. Figure 3 also shows the state diagram.

Illustrated in Figures 4 and 5 are the nonpipelined bus cycles and the transitions to and from pipelined bus cycles. Both the read and write cycles can be seen. The sequence of events for a typical nonpipelined cycle is:

1) Sampling the ADS# input to sense the beginning of a bus cycle while various bus controls and addresses are latched as needed at the end of the first clock.

2) At the end of the second or a later clock, the READY# input is sampled to sense the end of the current cycle. If READY# is sampled active, go to Step 1 on the next clock.

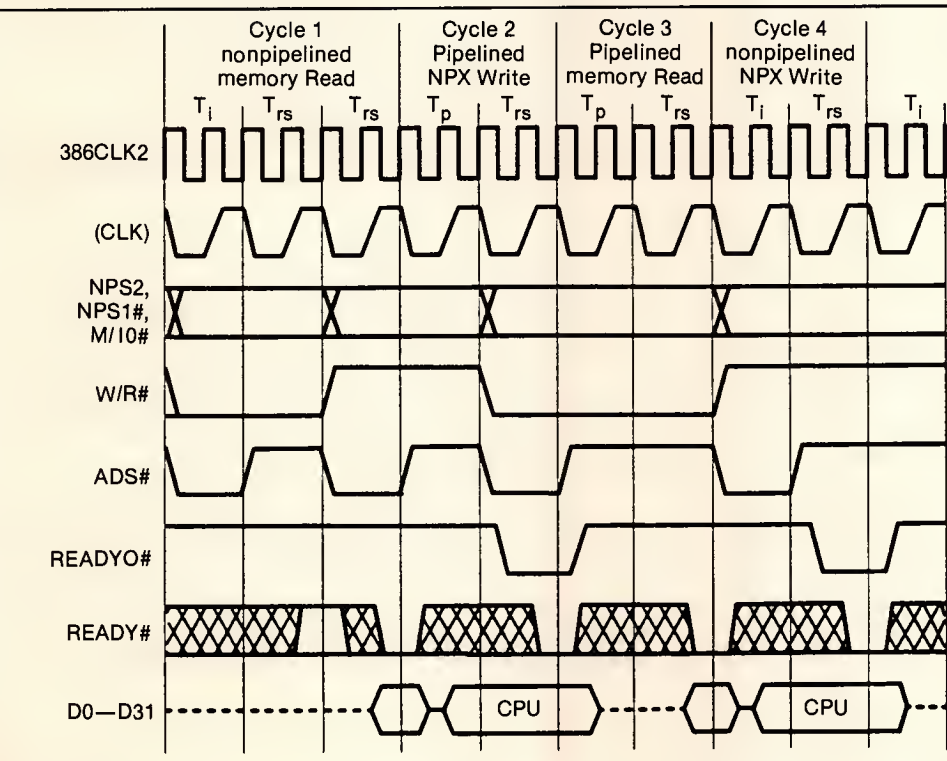
A pipelined cycle involves sampling the ADS# active again before READY# is found active to terminate the current cycle. The pipelined mode allows the next cycle to start as soon as the second clock of the pending cycle is entered.



Cycles 1 and 2 represent part of the operand transfer cycle for instructions involving either 4-byte or 8-byte operand loads.

Cycles 3 and 4 represent part of the operand transfer cycle for a store operation.

Cycles 1 and 2 could repeat here or T_i states for various nonoperand transfer cycles and overhead.



Cycle 1-Cycle 4 represent the operand transfer cycle for an instruction involving a transfer of two 32-bit loads in total. The opcode write cycles and other overhead are not shown.

Note that the next cycle will be a pipelined cycle if both READY# and ADS# are sampled active at the end of a T_{rs} state of the current cycle.

Figure 5. Fastest transitions to and from pipelined cycles.

Internal architecture

Figure 2 (shown earlier) illustrates the internal architecture of the 80387 coprocessor. The three major functional units in the 80387 are the:

- Bus Control Logic,
- Data Interface and Control, and
- Floating-Point Unit.

In general, the 80387 core is the combination of the Floating-Point Unit and the Data Interface and Control Unit. Bus Control Logic communicates solely with the CPU using I/O bus cycles, which are initiated automatically by the CPU when executing floating-point instructions. This unit relies on the CPU to perform all the memory cycles. Bus states are decoded to recognize data transfers to/from the 80386. All 80386 pipelined bus cycles are supported (refer to Figure 5).

The instruction decoder inside the Data Interface and Control Unit decodes the floating instructions to direct the dataflow in the FIFO. This unit generates the bus synchronization signals, BUSY#, PEREQ, and ERROR#, to reduce delay through the Bus Control Logic. It is also responsible for sequencing the microinstructions to control the Floating-Point Unit as well as the rest of the core.

The Floating-Point Unit executes all the instructions that involve the register stack, for example, the arithmetic, logical, transcendental, constant, and data transfer instructions. The data path in the unit is 84 bits wide (68 significant bits, 15 exponent bits, and a sign bit). This width allows internal operand transfers to be performed at very high speeds.

Figure 6 is the 80387 die photograph outlining the various units.

Features and major enhancements

The 80387 floating-point coprocessor features additional instructions not present in the 8087 and 80287 and an increased range of operands for several instructions.

New instructions. Two types of new instructions appear in the coprocessor. One type better supports the requirements and recommendations of the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.* Examples of such instructions are FPREM1, which supports a partial remainder as required by the standard, and FUCOM(P)(P), which performs unordered comparison. The other type of new instructions includes the trigonometric functions that are not already implemented in the 8087/80287, for example, FSIN (sine), FCOS (cosine), and FSINCOS (sine and cosine in one instruction). FPTAN (partial tangent) and FATAN (arc tangent) available on previous numeric processors are also supported.

With the new built-in sine and cosine functions, calculating the inverse of the trigonometric functions becomes easier and faster.

ASIN(Z) = ATAN2[Z,SQRT(1-Z^2)], when Z = SIN(Theta)

ACOS(Z) = ATAN2[SQRT(1-Z^2),Z], when Z = COS(Theta)

Note: The ATAN2 function uses two operands (Y and X) and computes the arc tangent, ATAN of Y/X; for example, ATAN2(y,x) = ATAN(y/x).

Wider operand ranges. In the 8087/80287 some instructions restrict operand ranges. Such restrictions require application software to verify that operands are in range before performing a desired operation. The 80387 widens the range for most of these limited instructions. For example, the arc tangent function is unlimited in operand range while tangent, sine, and cosine are now almost unlimited in range (2^{63}). For large operands, the range improvement not only reduces the programming overhead mentioned above but also improves accuracy. More accuracy results because large operands are reduced with a more-precise, built-in PI (in 84-bit internal extended format versus 80-bit extended format for external PI). Since the trigonometric functions are periodic, for very large operands, the error accumulated as a result of using a less-precise PI value can result in significant accuracy degradation as illustrated here.

Let PI be the external PI and
MPI be the PI available to the machine performing the reduction.
Let E = PI – MPI.

If SIN(N*PI + Theta) needs to be calculated, the accumulated error will be:
(for the ease of calculation, let N be an even number)

SIN(N*PI + Theta) = SIN[N*(MPI + E) + Theta)
= SIN(N*E + Theta)
= SIN(Theta)*[SIN(NE)*COT(Theta) + COS(NE)]
Relative_Error =
[SIN(NE)*COT(Theta) + COS(NE)]

It is obvious that for a large enough N, the computed result is very inaccurate.

Another enhancement permits the 80387 to support Denormal as an operand in many other instructions that might otherwise not be supported by the 8087/80287.

Applications

The compatibility of both the 80386 and the 80387 at the object-code level allows existing application software to run on the system without modification.

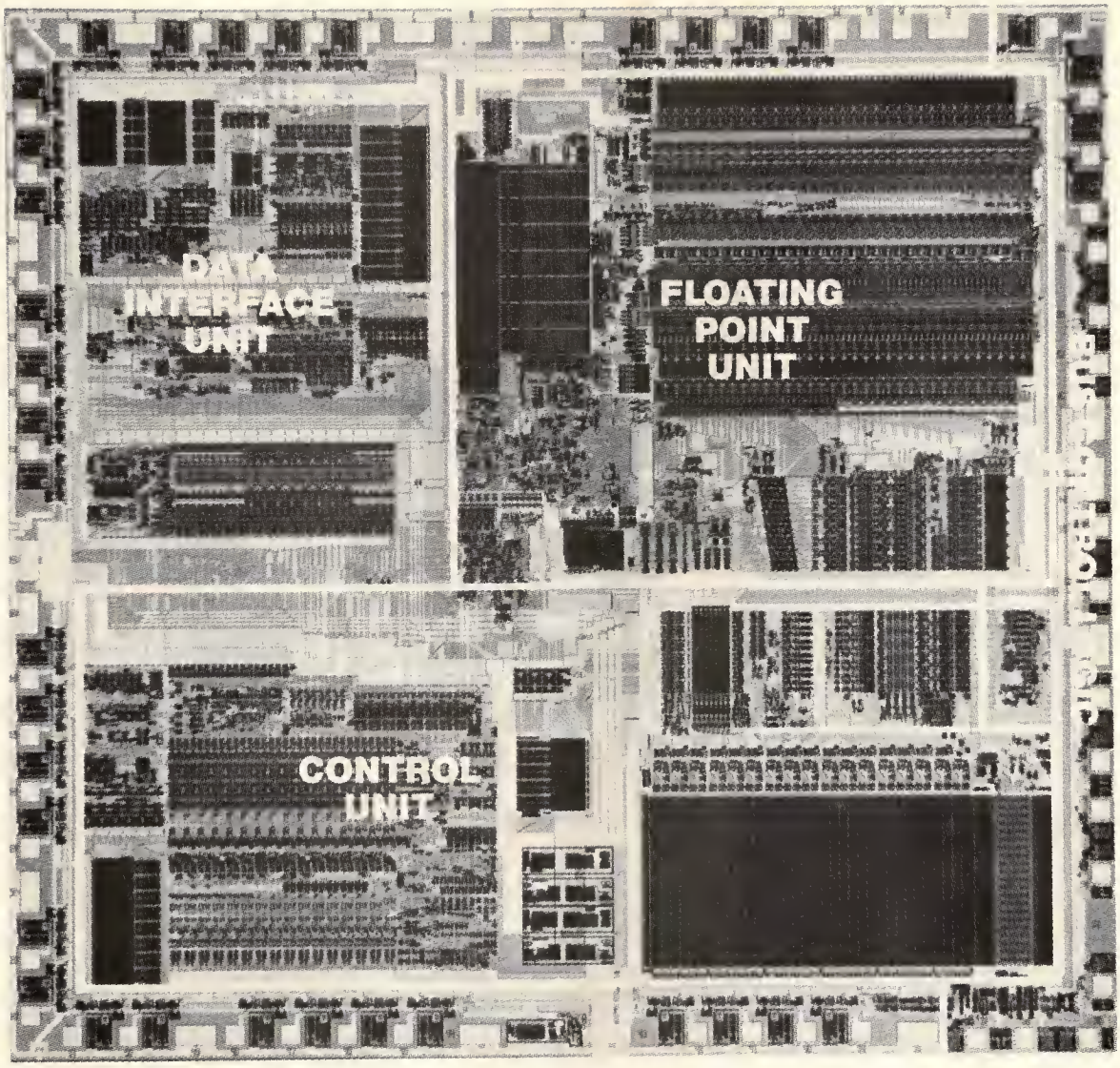


Figure 6. 80387 die photograph.

Established uses of the 8087 and 80287 include math accelerators in personal computers, multiuser microsystems, and multiprocessing systems such as the iPSC Intel Personal Supercomputer.

The 80387's speed improvement of four to six times over the 80287 extends math applications to such new areas as workstations and real-time and embedded systems. Examples of real-time and embedded applications are process control, robotic, navigational, and guidance systems.

Typical hardware configuration. The 80386/80387 interface shown in Figure 1 represents the simplest interface. One key feature permits the CKM input to the 80387 to be strapped low so the 80387 can run in the asynchronous mode of operation. When running asynchronously, the 80387 core can run at a different speed than the system interface. In this case, an external oscillator meeting the AC timing requirements for the 387CLK2 input is needed. This mode of operation offers design flexibility during the design phase and numerics performance upgrading in the future without having to redesign the overall system interface.

This interface is typically used in most of the 80386-based systems (such as those to be described shortly). PC AT-compatible systems use a slightly different 80386/80387 interface as described later.

Single-board computers. A single-board computer using the 80386/80387 with on-board cache and/or a large amount of static RAM, DMA controller, interrupt controller, and timers could be used in stand-alone or embedded systems. Stand-alone systems include workstations and expert systems; embedded systems could be used for process control, robotics, navigation, and phototypesetting purposes. The 80387 is particularly useful in embedded systems with positional and rotational matrix calculations involving a lot of accurate floating-point computations. The 84-bit internal computing hardware provides the needed accuracy for recursive computations and precise control of distance, position, and motion. The built-in FSIN, FCOS, and FSINCOS functions (generating both the sine and cosine) significantly reduce computing time. This feature is especially important for complex robotic applications.

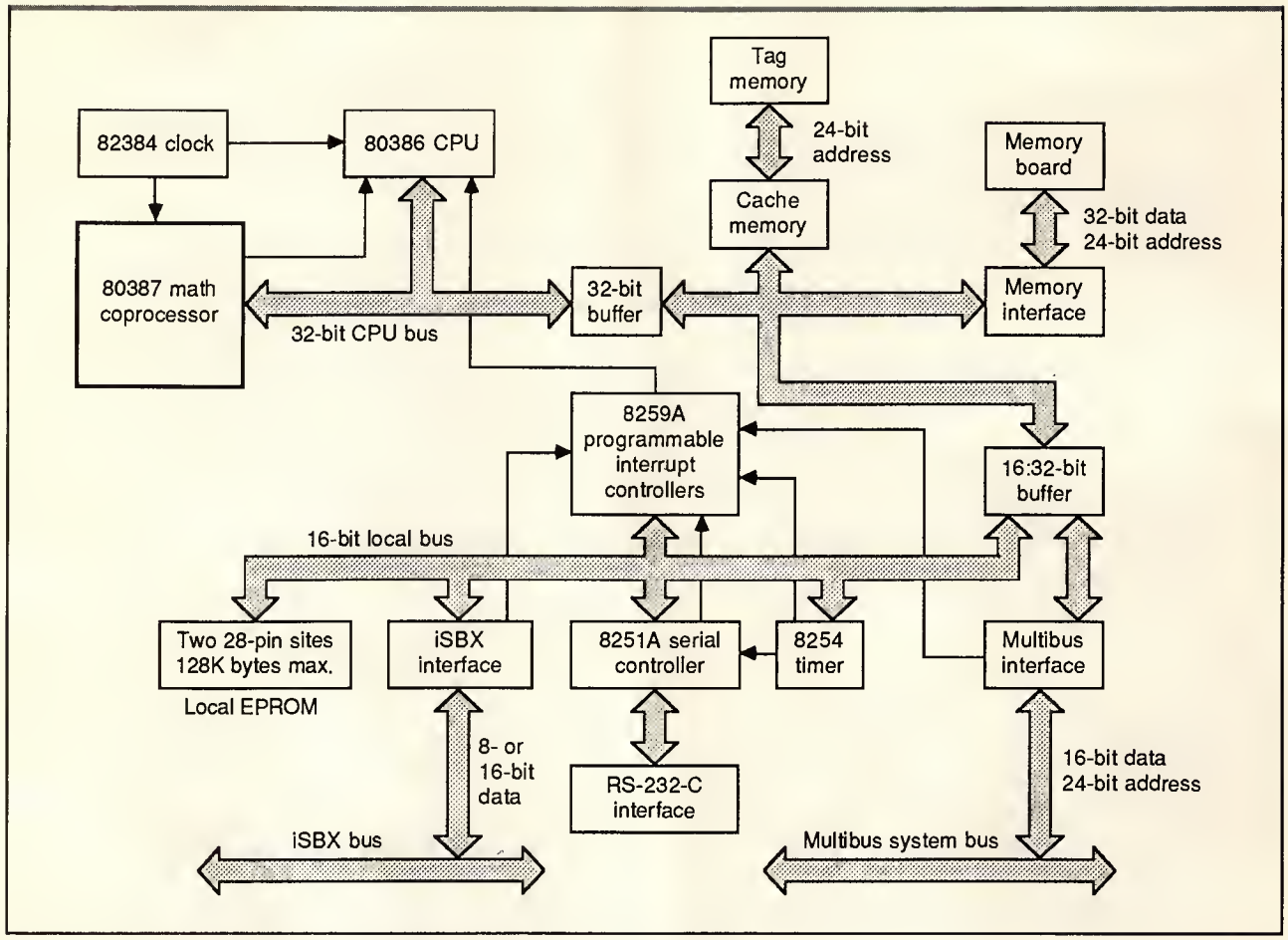


Figure 7. Block diagram of the single-board computer, the iSBC386/20.

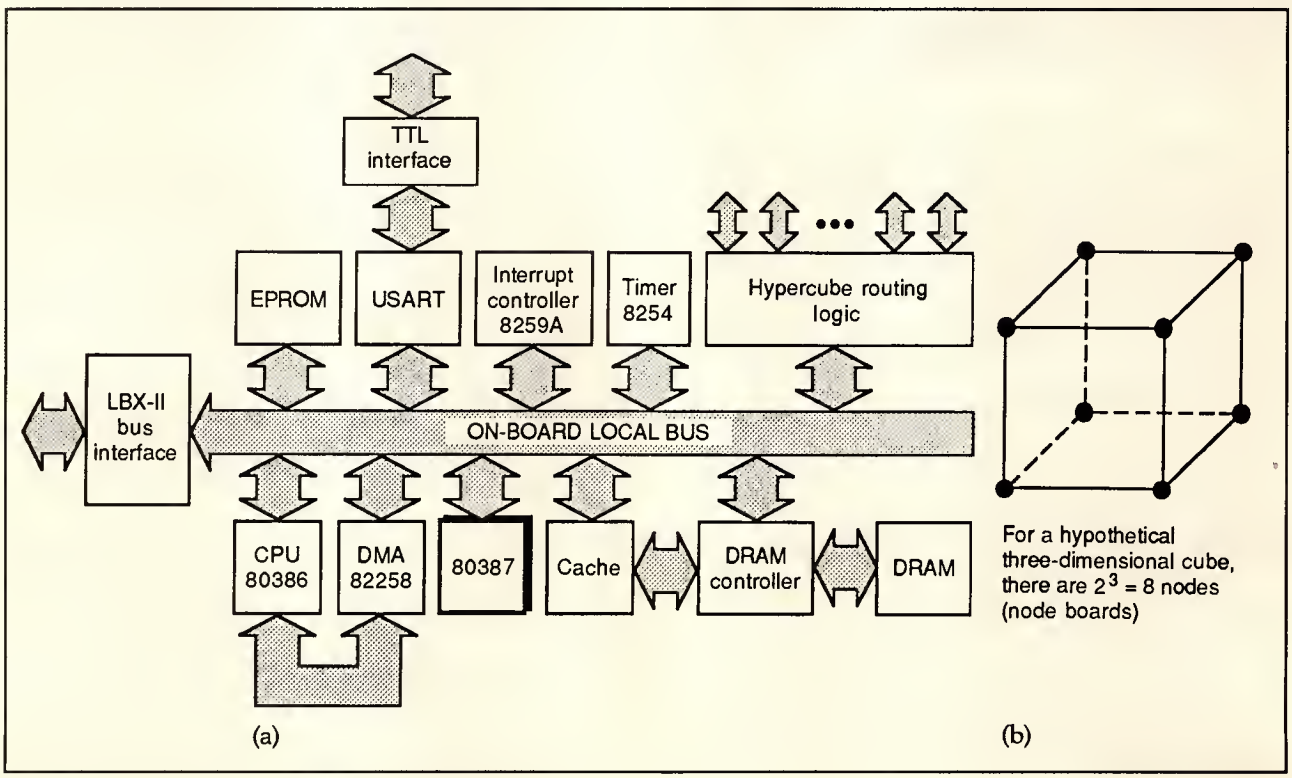


Figure 8. Node board block diagram (a) and hypercube structure (b).

A block diagram of a typical 80386/80387, single-board iSBC386/20 computer appears in Figure 7. The board contains 64K bytes of cache and executes with less than one wait state. It serves as the CPU board in various engineering workstations and embedded systems. The system is expandable in hardware configuration using the Multibus and various local bus extensions. Major system software is targeted around this board.

Multiprocessing systems. In a hypercube multiprocessor system, such as the iPSC Personal Supercomputer, each node in the cube is an independent, single-board computer. The node can contain an 80386/80387 pair and some dual-port local memory for initialization, kernel loading, interprocess communication, and user programming. The two key factors in the design are the computing power of each node and the efficiency of interprocessor communication. The needed computing power is provided by the 80386/80387, while dedicated communication hardware supports the low-level internode message routing. The use of 80386/80387 reduces the cost of each node, increases nodal performance, and allows the use of readily available software development tools and languages. Existing system software for the 80286/80287-based iPSC computer can be run on the new system. The 80387 floating-point coprocessor greatly increases the numeric processing power.

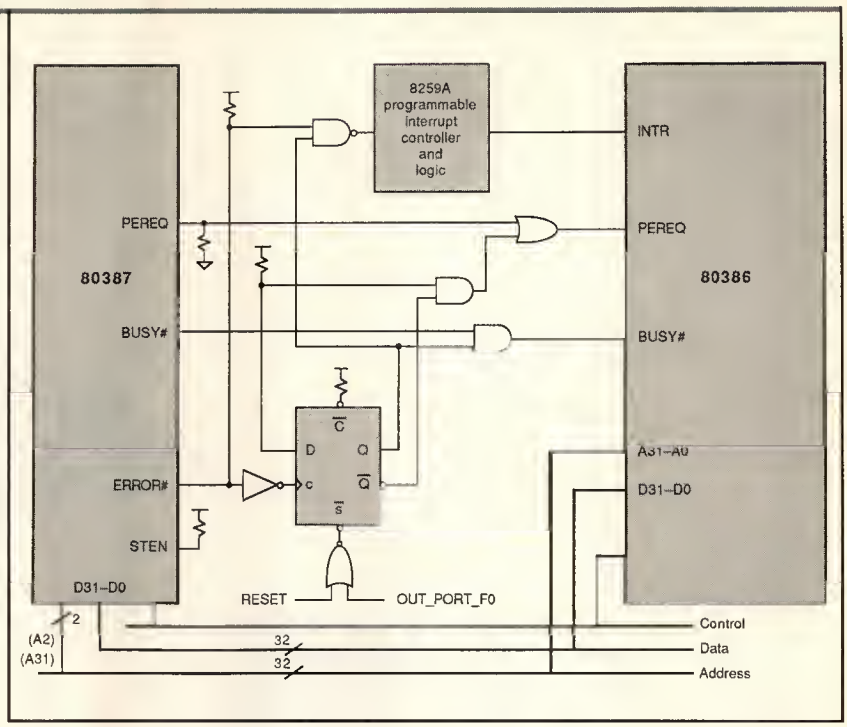


Figure 9. Block diagram of a conceptual IBM PC AT-compatible 80386/80387 interface.

Each of the iPSC processing nodes is connected to a host processor called the Cube Manager, which provides programming and system management. The Cube Manager allows the iPSC system to operate as a stand-alone system or a computational server within a distributed processing environment. It could be another 80386-based microcomputer system. A block diagram of a node board can be seen in Figure 8.

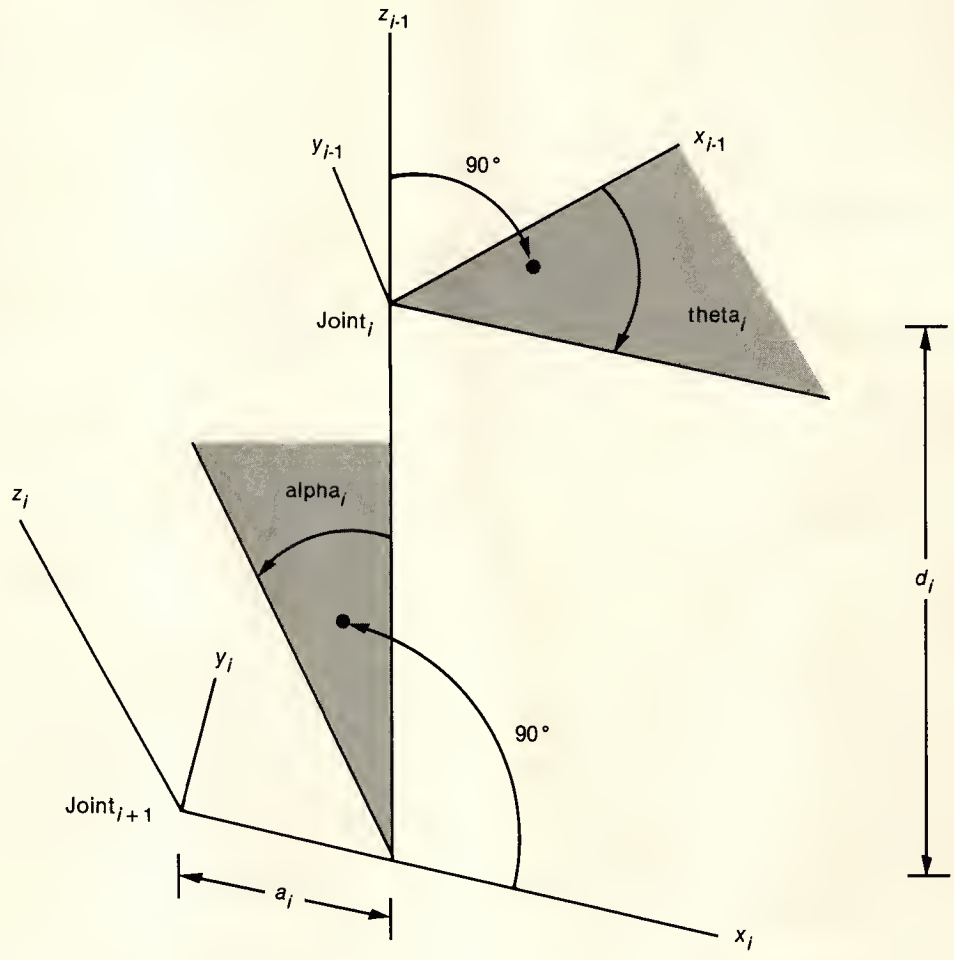
The diagram shows that the 80386 and 80387 are closely tied together as a single piece of hardware integrated into a multiprocessor system. The 80386 LOCK instruction and atomic page table updates allow multiple CPUs to share data and page tables.

Distributed processing. A distributed process control system consists of many distributed nodes controlling the different processing functions in a factory. Functions such as temperature sensing and controlling, valve opening and closing, and stepper motor control require local intelligence. A complex system can contain over a hundred nodes or drops.

Each node contains the central processing module, which may be implemented with an 80386/80387 system such as an iSBC386/20 board or a distributed controller. The 80387 offers fast, real-time floating calculations to effect environmental and robotic control. Application software for existing 8086/8087- and 80286/80287-based distributed systems runs directly on the new 80386/80387-based distributed systems. This capability significantly reduces software development cost and time.

IBM PC AT-compatible interface. Just as the PC AT personal computer has become a versatile system for a variety of applications, the use of an 80387 in an 80386-based personal computer could significantly improve the numeric performance, opening up many new applications. For example, the 80387 could accelerate signal processing to the performance range of many existing array-processor-based PC accelerator cards, which could cost as much as 30 times that of an 8087. In addition, the 80386/80387 could be used in artificial intelligence or other accelerator add-on cards on a PC AT platform. For computation-intensive systems, the 80387 should be configured to run in asynchronous mode. In this case, the internal number crunching could be clocked at a faster rate than that of the bus interface. This cuts down the cost of having to design and implement a faster overall system.

Conceptually depicted in Figure 9 is the IBM PC AT-compatible 80386/80387 interface. The external interface logic serves as error-signaling hardware. In this configuration, the ERROR# output of the 80387 is not connected to the 80386 directly but rather to interrupt logic (8259A), which in turn drives the INTR input of the 80386. The use of interrupts for math-



$A(i\text{-}1,i) =$

COS theta$_i$	-COS alpha$_i$ SIN theta$_i$	SIN alpha$_i$ SIN theta$_i$	A_i * COS theta$_i$
SIN theta$_i$	COS alpha$_i$ COS theta$_i$	-SIN alpha$_i$ COS theta$_i$	A_i * SIN theta$_i$
0	SIN alpha$_i$	COS alpha$_i$	d_i
0	0	0	1

where

theta$_i$ = The angular displacement of the x_i axis from the x_{i-1} axis by rotating around the z_{i-1} axis (anticlockwise)

d_i = The distance from the origin of the $(i\text{-}1)$th coordinate system along the z_{i-1} axis to the x_i axis

a_i = The distance of the origin of the ith coordinate system from the z_{i-1} axis along the $-x_i$ axis

alpha$_i$ = The angular displacement of the z_i axis from the z_{i-1} about the x_i axis (anticlockwise)

Figure 10. Coordinate relationships between adjacent joints.

error signaling is a technique that was carried over from the design of the 8088-based personal computer. The 8088 contains no ERROR# pin, and hence INTR signals math error. To be able to run existing

Code Segment of the Denavit-Hartenberg Transformation Matrix

```
TRANS_PROC proc far


   ; Calculate alpha and theta in radians
   ; from their values in degrees
     fldpi
     fdiv    d180

   ; Duplicate pi/180
     fld     st

     fmul    qword ptr ALPHA_DEG[ecx*8]
     fxch    st(1)
     fmul    qword ptr THETA_DEG[ecx*8]

   ; theta(radians) in ST and
   ; alpha(radians) in ST(1)

   ; Calculate matrix elements
   ; a11 = cos theta
   ; a12 = - cos alpha * sin thet
   ; a13 = sin alpha * sin theta
   ; a14 = A * cos theta
   ; a21 = sin theta
   ; a22 = cos alpha * cos theta
   ; a23 = -sin alpha * cos theta
   ; a24 = A * sin theta
   ; a32 = sin alpha
   ; a33 = cos alpha
   ; a34 = D
   ; a31 = a41 = a42 = a43 = 0.0
   ; a44 =1

   ; ebx contains the offset for the matrix

     fsincos           ;cos theta in ST
                       ;sin theta in ST(1)
     fld     st        ;duplicate cos theta
     fst     [ebx].a11 ;cos theta in a11
     fmul    qword ptr A_VECTOR[ecx*8]
     fstp    [ebx].a14 ;A * cos thetain a14
     fxch    st(1)     ;sin theta in ST
     fst     [ebx].a21 ;sin theta in a21
     fld     st        ;duplicate sin theta
```

PC software, the PC AT also signals math error via interrupts, although the 80286 used in the PC AT also has the same ERROR# as the 80387. This explains the need for the additional hardware to simulate the error-signaling hardware in the PC AT.

While an error is signaling through the interrupt logic, the CPU must be prevented from passing another math instruction to the 80387. The error latch, as triggered by the low-going edge of the 80387 ERROR# output, serves to extend the active time of the BUSY# signal to the CPU. The PEREQ is reactivated when ERROR# occurs. This action brings the CPU out of any wait state for pending data transfer, if any, to complete. The error latch is cleared once the CPU starts executing the interrupt service routine (via the signal OUT_PORT_F0).

Although this configuration does not take full advantage of the most-efficient-possible interface between the 80386 and 80387, it provides a PC AT-compatible interface for the personal computers. A similar interface can be found in the new IBM PS/2 systems.

Numerics programming example

Numerics programming can be illustrated with the following example showing the way some common 80387 instructions are used. The new, built-in FSINCOS function calculates both the sine and cosine values for the given angle in radians, saving hundreds of clocks in the computing time. The possibility for parallel execution of the 80386 and 80387 instructions is also highlighted. The parallel operations balance and improve overall system performance.

Consider the calculations of matrix elements for the Denavit-Hartenberg transformation matrix used to model the kinematics of a robot arm. Shown in Figure 10 are the local coordinate systems originated at adjacent joints of a robot arm. The matrix $A(i\text{-}1,i)$ transforms the ith coordinate system to the $(i\text{-}1)$th coordinate system.

The segment of the code that calculates the matrix elements is shown in the box above. Although only 64-bit memory operands are used, the internal operations are carried out in the 80-bit extended format.

```
fmul    qword ptr A_VECTOR[ecx*8]
fstp    [ebx].a24 ;A * sin theta in a24
fld     st(2)     ;alpha in ST
fsincos           ;cos alpha in ST
                  ;sin alpha in ST(1)
                  ;sin theta in ST(2)
                  ;cos theta in ST(3)
fst     [ebx].a33 ;cos alpha in a33
fxch    st(1)     ;sin alpha in ST
fst     [ebx].a32 ;sin alpha in a32
fld     ST(2)     ;sin theta in ST
                  ;sin alpha in ST(1)
fmul    st,st(1)  ;sin alpha * sin theta
fstp    [ebx].a13 ;stored in a13
fmul    st,st(3)  ;cos theta * sin alpha
fchs              ;-cos theta * sin alpha
fstp    [ebx].a23 ;stored in a23
fld     st(2)     ;cos theta in ST
                  ;cos alpha in ST(1)
                  ;sin theta in ST(2)
                  ;cos theta in ST(3)
fmul    st,st(1)  ;cos theta * cos alpha
fstp    [ebx].a22 ;stored in a22
fmul    st,st(1)  ;cos alpha * sin theta
;
; To take advantage of parallel operations
; between the CPU and NPX
;
push    eax  ; save eax
;
; also move D into a34 in a faster way
mov     eax, dword ptr D_VECTOR[ecx*8]
mov     dword ptr [ebx + 88], eax
mov     eax, dword ptr D_VECTOR[ecx*8 + 4]
mov     dword ptr [ebx + 92], eax
pop     eax  ; restore eax
fchs              ;-cos alpha * sin theta
fstp    [ebx].a12 ;stored in a12
                  ;and all nonzero elements
                  ;have been calculated
ret

TRANS_PROC endp
```

The 80387 fully complies with the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic* in addition to being fully compatible with the previous generations of floating-point processors. The CPU/coprocessor interface has been improved, especially for operations that involve 32/64-bit memory operands. For those instructions, the CPU can start writing the new opcode into the coprocessor before the 80387 finishes executing its current instruction. The asynchronous mode of the 80387 allows flexibility in design and future upgrading.

The improvement in performance of four to six times over the 80287 opens up many new applications for the 80387, especially in real-time, embedded systems, which require fast and high-precision numerics processing. For existing applications, the fast 80387 speeds up existing software and makes other time-consuming software feasible.

Additional reading

A Proposed Radix and Word Length Independent Standard for Floating-Point Arithmetic, (Draft 1.0 of IEEE task P854), *IEEE Micro,* Aug. 1984, pp. 86-100.

ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, Inst. Electrical and Electronics Engineers, New York, 1985.

Denavit, J., and R. Hartenberg, "A Kinetic Notation for Lower-Pair Mechanisms Based on Matrices," *J. Applied Mechanics,* June 1955, pp. 215-221.

8087 and 80287 Programmer's Reference Manuals, Intel Corporation, Santa Clara, Calif., 1985.

80386 Programmer's Reference Manual, Hardware Reference Manual and Datasheet, Intel Corporation, 1986.

80387 Datasheet and *80387 Programmer's Reference Manual,* Intel Corporation, 1987.

Lee, C.G., "Robot Arm Kinematics, Dynamics, and Control," *Computer*, Dec. 1982, pp. 63-80.

Nave, R., "Implementation of Transcendental Functions on a Numeric Processor," *Microprocessing and Microprogramming,* 1983, pp. 221-225.

Palmer, J., et al. "Making Mainframe Mathematics Accessible to Minicomputers," *Electronics*, May 1980, pp. 114-121.

Technical Reference Manual for Personal Computer AT, IBM Corporation, Boca Raton, FL 33432, 1985.

David Perlmutter is a senior component design engineer at Intel Design Center in Haifa, Israel. His responsibilities include architecture definition and project management of VLSI designs. He has served as the project manager for the 80387 and previously as the design engineering liaison for the Israeli and Santa Clara design centers. Upon joining Intel Israel in 1980 he worked with the design and definition of memory controllers.

Perlmutter received the BSEE degree from the Technion-Israel Institute of Technology, Haifa, in 1980.

Alan Kin-Wah Yuen has worked on a number of VLSI design projects since joining the Microprocessor Division of Intel Corporation in 1982. As a senior design engineer, he was involved with the design of the 32-bit microprocessor, the 80386. He is currently working with the technical marketing group of the Microprocessor Division. His interests include computer system and VLSI designs.

Yuen received the BSEE degree from Rensselaer Polytechnic Institute in 1981 and the MSEE and MBA degrees from the University of California, Berkeley, in 1982 and 1987.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 156 **Medium** 157 **Low** 158

Wondering where to get back issues?

IEEE MICRO

Contact IEEE Computer Society, PO Box 80452, Worldway Postal Center, Los Angeles, CA 90080 for 1984 and 1985 issues of *IEEE Micro*.

Special Offer
$3.00 per copy/
$15.00 minimum order

It's time to renew your subscription to *IEEE Micro*.

STOP

IEEE/CS members: Remember to return your IEEE dues notice, and we'll see that you continue to receive *IEEE Micro* throughout 1988.

engineers

Software Engineers

Northrop Corporation's Defense Systems Division in Rolling Meadows, Illinois is the fastest-growing enterprise in an expanding electronic countermeasures industry. True breakthroughs in technology are born of a success system in an environment of cooperation and teamwork among individuals who have the opportunity to test and refine their ideas. Northrop offers professionals with a **BSCS, BSEE, BS Math or Physics (or equivalent) MS preferred, and a minimum of 3 years experience,** opportunities in the following areas. **Management, System Architect, Technical Leaders and engineering assignments available.**

System Programmers

Our many and varied applications require significant growth in our support capabilities. To facilitate our development efforts, we need the best people with experience in:

- LANGUAGES, including Ada, Assembler, C, FORTRAN, JOVIAL, and Pascal
- OPERATING SYSTEMS, including UNIX and VMS
- Development of Software Tools
- Performance Modeling and Evaluation
- Use of Software Structured Development Methodologies
- Development of Real-Time Operating Systems

Software Systems Engineers

Our software engineers are creative designers. They develop software from system requirements through implementation. We're looking for talented people with experience in:

- Software Requirements Analysis
- Architectural Design
- Performance Specification and Modeling
- Interface Design and Specification
- Software Validation and Test Specification

ECM/EW Systems Software Engineers

ECM/EW Systems are our business, and we offer significant challenge and responsibility to the best people with experience in:

- Real-Time Control Systems
- Embedded Computer Systems
- Radar Data Processing
- System and Unit Level Diagnostics
- Object Discrimination and Classification
- ECM Algorithm Development
- Kolman Filtering
- Optimal Control

Hardware Diagnostics Software Engineers

We design and develop advanced systems using the latest hardware and software technologies, and high-speed architectural concepts for our military clients. We seek people with experience in:

- Intelligent Control Panel Systems
- Development
- Built-in-Test
- Functional Test
- Micro and Macro Diagnostics for Fault Identification

Artificial Intelligence

Artificial intelligence technologies promise state-of-the-art solutions to complex ECM/EW challenges. Positions require people who can bring AI technologies to avionic electronics, with experience in:

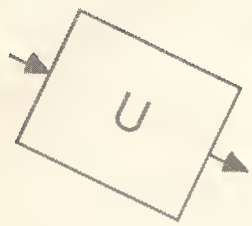
- LANGUAGES, including: Ada, C, LISP, and Prolog
- Knowledge Engineering
- Expert Systems Development
- System Prototyping
- Implementation of AI Technology in Real-Time Embedded Systems

Northrop offers a highly competitive salary commensurate with level of experience plus a comprehensive benefits portfolio. Interested individuals are encouraged to forward resume to: **James Frascona, Technical Recruiter, Dept. C98, Northrop Corporation, Defense Systems Division, 600 Hicks Road, Rolling Meadows, IL 60008.** We are an equal opportunity employer M/F/V/H. U.S. Citizenship may be required for certain positions. ***ONLY PRINCIPALS NEED APPLY.***

NORTHROP

Defense Systems Division
Electronics Systems Group

VLSI AND SYSTEM PERFORMANCE MODELING



Complex VLSI chips and systems call for performance evaluation techniques that are easy to understand, change, and maintain. High-level simulation environments answer this need.

Sorin Iacobovici, ChakChung Ng
National Semiconductor

Due to advances in semiconductor technology, a VLSI chip carries hundreds of thousands of transistors. As a result, the processor and peripheral chips become more complex. This complexity makes the selection of the internal architecture that will give the best performance a formidable task.

At the same time more computer systems are using multiprocessors or parallel processor systems to achieve mainframe performance at a fraction of the cost of a mainframe. These systems use multiple processors, usually with a dedicated cache for each processor, and share global memory and I/O. Because of contention between the processors for the shared resources, the system performance for a given work load and the performance increase from adding extra resources (processors, buses) are hard to quantize. (The term *work load* designates the processing requests to the computer system.)

For these complex chips and systems, the product definition phase cannot be completed without simulating different configurations for the desired work loads and comparing their performance. Because of the strong dependence of the computer system performance on the work load, the performance comparison between different system configurations should be calculated for the same work load. In the case of processor chips, performance can be simulated in either stand-alone or system environments.

To simulate the performance of a particular chip or system configuration, the designer must provide data concerning the performance of the active resources, that is, those modules providing service to a job. The simulation output will show not only the overall performance but also the bottlenecks that limit this performance. By analyzing this data, the chip or system architect can improve the chip/system configuration. The chip/system designer also can identify the most critical modules for the system perfor-

0272-1732/87/0800-0059$01.00 © 1987 IEEE

Table 1.
Characteristics considered important to our simulation environment.

Characteristic	Comments
Graphic input	Represents any system as an information flow graph. Ideally, the model should be written by simply declaring the characteristics of each model element and the model topology.
Very high level primitives	Provides very high level programming constructs for specifying objects and actions found in models. Provides hooks so model programmers can build their own model elements.
Structured form	Enables the system representation as a hierarchy of submodels to allow top-down modeling.
Versatility	Contains all the necessary elements to simulate the targeted range of systems (hardware and software). Capable of working with either statistical or trace data.
Extensive statistics gathering	Provides all the information necessary for analyzing the throughput and latency of the whole system, as well as the behavior of each element of the system.
Graphic output	Provides graphic viewing of the performance measures of the simulation. Also desirable is an animation capability to display the movement of jobs on the model diagram and the accompanying changes in performance measures.
Portability	Runs on a variety of computers and operating systems.
Edit and run speed	Offers fast model turnaround time and fast model compilation and simulation run.
Cost	Is inexpensive to buy and maintain.

mance and optimize them during the design phase, while using a more conservative design for the noncritical modules.

Some of the assumptions made during the product definition phase and used in the simulation model might change during the design phase. The simulation model might then be used to evaluate the impact of the proposed change(s) on performance and identify the bottlenecks (the critical modules) in the new configuration.

Choosing a simulation environment

Choosing the right performance simulation environment from the alternatives currently available heavily influences (1) the model's readability, portability, and maintainability, (2) the programmer's productivity, and therefore (3) the model's usefulness. Though not an exhaustive list, Table 1 lists some of the most important elements considered when we chose our simulation environment.

General-purpose simulation languages like GPSS, Simscript, and Simula have existed for quite some time. Many simulation models have also been written in general-purpose programming languages like C, Pascal, and Fortran. These languages, though, are not high-level languages from the simulation point of view. The simulation model of a complex system, if written in one of these languages, takes considerable effort to write and debug. Such a model is also very hard to maintain and modify.

Even simulation models written in modern simulation environments like Endot and Verilog are still hard to use and maintain if the system is complex. These simulation packages, though, allow not only performance modeling of a system but also simulation of the same system at lower levels, like the register-transfer level and gate level, as part of the design process.

Special-purpose performance modeling environments, offered recently by several companies, can dramatically boost the programmer's productivity and enhance the model's maintainability. Based on a methodology using pictorial representation of models, languages like Resq, or Research Queueing Package, from IBM or PAWS, the Performance Analyst's Workbench System, from IRA provide a rich set of very high level modeling elements.[1,2] These modeling elements represent abstractions of concepts used in computer systems, so these languages are easy to learn and use. For performance modeling purposes, the Series 32000 architecture group from National Semiconductor chose the PAWS simulation language (see the box on "PAWS Meets Simulation Modeling Requirements"). PAWS is a simple and powerful high-level simulation language that was installed to run in a Unix environment.

PAWS Meets Simulation Modeling Requirements

Most lower level simulation languages are procedural: The programmer has to code detailed algorithms describing the behavior of each model resource for each type of event. Unlike these languages, the Performance Analysts' Workbench System is a declarative language. In PAWS the programmer simply declares the characteristics of the model primitives, be it the *nodes*, where the information is processed, or the model *topology*, which describes the information flow through the model.

Building a PAWS model starts with a pictorial representation of the system (graphic input) using the very high level PAWS simulation primitives. The system model-building task is further simplified by using the GPSM, or Graphical Programming of Simulation Models, package. Each PAWS node is represented in GPSM by a symbol, or icon, used to build the information flow graph for the model, and by a template, used to specify, or declare, the properties of each node. Most computer systems can be modeled by using the PAWS high-level primitives only. If special functions are needed, *User* nodes and *Compute* nodes can be used to implement these functions.

The *transactions* represent in PAWS the unit of information processed by the simulated system. Different types of transactions are differentiated: a *category* and a *phase* are associated with each transaction. A category is a name associated with each transaction; it does not change during the transaction's lifetime. The phase of the transaction is an integer number that may be changed as the transaction progresses through the model. Both the information routing through the model and the information processing at every node are functions of the nature of the information (that is, of the transaction's category and phase).

The nodes of the simulation model process each transaction as programmed. PAWS provides node types that satisfy virtually all modeling needs. These nodes and the symbols used to represent them are presented in Figure A. The *resource management* nodes, for instance, represent the resources of the modeled system. Since transactions might have to wait for these resources, these nodes have *queues* associated with them.

The resource management nodes can be active or passive. The active nodes are servers, simulated

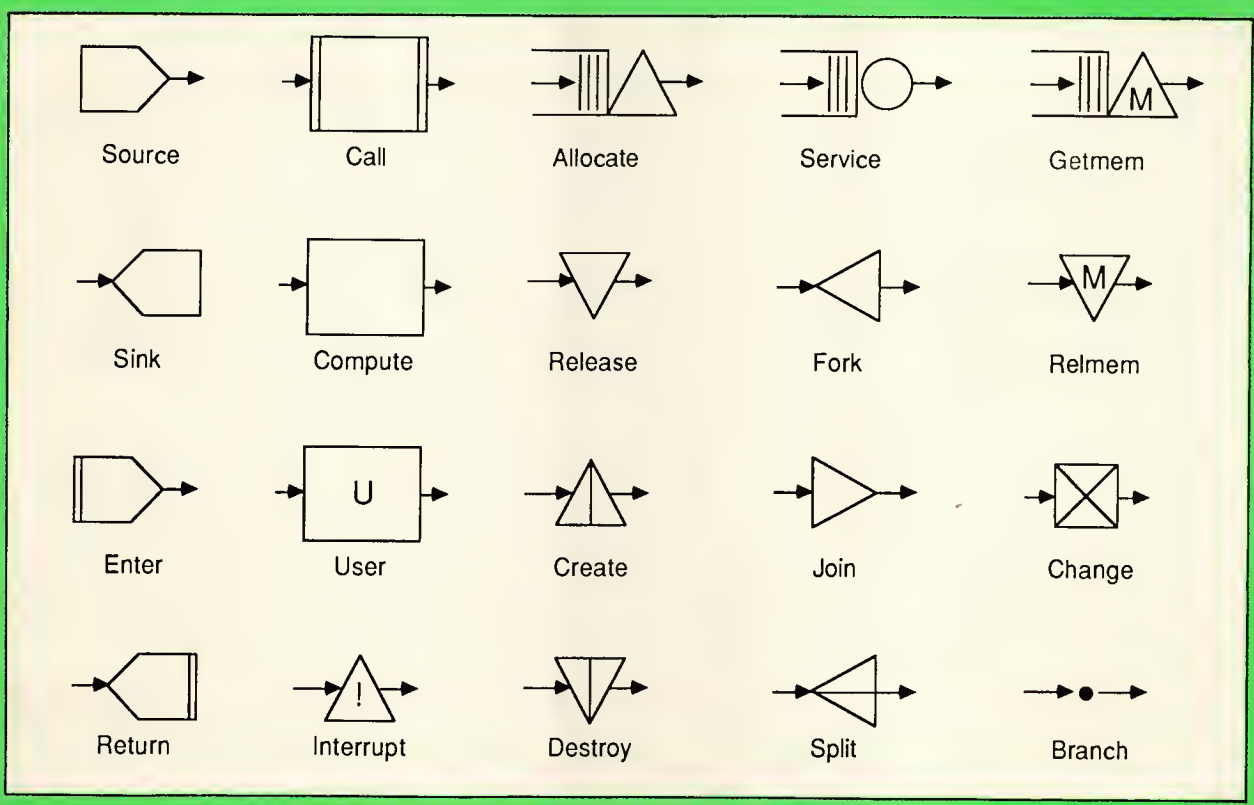


Figure A. Types and symbols for PAWS nodes.

Simulating a computer system

Before the performance modeling of a computer system can begin, the designers have to decide not only on the architecture to be modeled but also on the input to the model and the type of information (the output) expected from the model.

The computer system performance is evaluated while running a certain work load. The work-load model, which generates the inputs to the computer system model, determines if the simulation is statistical (probabilistic) or deterministic.[3,4]

A deterministic computer system simulation can be accomplished by using a work-load trace as input to the computer system model. Such a trace could include executed and prefetched instructions, interruptions (external interrupts and exceptions), and DMA activity. Though more difficult to feed into a model (in PAWS, for instance, a User node, involving some C or Fortran user-written code, is necessary), the trace input allows a simple, precise model of the system to be constructed.

The code in this case is that run by the actual sys-

in PAWS by *Service* nodes. The passive nodes represent resources that do not perform work, but must be possessed by a transaction to operate (for example, buses). A passive resource is usually represented in a PAWS model by *tokens*. A transaction acquires the resource (the tokens) at an *Allocate* node and releases it at a *Release* node. *Create* and *Destroy* nodes can also be used to handle the tokens. Memory is another type of passive resource and can be allocated by the *Getmem* node in a similar way as the Allocate node and released by the *Relmem* node.

Another type of node, the *routing* node, can be used to create and destroy transactions, as well as to alter transaction flow through the system. PAWS features six routing nodes: *Source, Sink, Split, Fork, Join,* and *Branch*.

Each model uses *simulation* variables, handled automatically by PAWS, and *user-defined* variables. The user-defined variables can be integer, real, or Boolean variables that are global (accessible by all transactions in all submodels) or local to the submodel in which the variables are defined (accessible by all transactions only in this submodel). The user-defined variables, as well as most of the simulation variables (for example, the transaction phase), can be accessed and modified in PAWS at Compute nodes or in the *Run* section of the model. If only the transaction's phase is to be changed, the *Change* node can be used instead of the Compute node.

Interrupt nodes can be used by one transaction to interrupt the processing of another transaction in the model, while User nodes allow the model to invoke subroutines written by the user in Fortran and C languages, making PAWS extensible.

Beginning with Version 3.0, the PAWS language makes possible a structured model design by supporting the concept of *submodels*. The submodels are implemented in PAWS by using *Call, Enter,* and *Return* nodes. System models can be built using a top-down approach (a hierarchy of submodels), which results in models simpler to design, understand, and maintain. If model modifications are necessary, either because of design changes or because a higher level of detail in the simulation is desired, only the affected submodels need to be modified.

A submodel can be invoked (called) by multiple higher level submodels. These requests will be queued inside this submodel according to the simulated queuing discipline and serviced as programmed, simulating in a simple and elegant way multiple system modules competing for the same resource.

Both the very high level simulation primitives and the submodel concept make a model written in PAWS easy to modify. After the modification of the information flow graph and of the node and topology declarations, the model is simply recompiled and run.

The PAWS output appears in several statistics files, which are the result of the simulation run. A *statistics report* feature allows the programmer to request histograms of specific queue lengths, queueing times, and response times (time needed by a transaction to get from a specified *Start* node in the model to a specified *End* node).

One of the weaker points of PAWS is the lack of graphic output. According to Information Research Associates of Austin, Texas, which developed and markets PAWS, the next PAWS revision will have this feature. Today, the programmer can generate in PAWS plots of performance measures as a function of different parameters. The programmer does this by expressing the chosen parameters as global variables in the simulation model and running simulations with the desired values for these parameters. The value of the global variables can be changed in the Run section of the PAWS program.

Future PAWS releases are planned to provide an animation feature. Presently, the *trace* flag in PAWS allows the generation of a trace file, which contains a brief description for each event simulated while the trace flag is turned on.

tem. This is important if the purpose of the system is to get very precise performance numbers, as the system interlocks depend on the instruction sequences executed. At the same time the system model is relatively simple. When simulating a CPU, for instance, only the type of the executed instructions and the addressing modes are relevant for the performance evaluation. Details such as new values of registers and memory locations are irrelevant in this case. This control-flow simulation, rather than full computational emulation, greatly simplifies the model.

Work-load traces can be obtained using either special software or special systems, like National's Megatracer.[5] Software tracers tend to be slow and trace only events related to the CPU. Their advantage is low cost, a result of their running on the traced system itself. Hardware tracers are much faster; they trace the system running the work load in real time, in a noninvasive manner. Both CPU-related events (executed instructions) and system-related events (interrupts and DMA accesses) can be traced by such a hardware tracer. The traced data can be filtered to serve as input to different simulations (performance modeling and cache simulation among others).

A statistical (probabilistic) simulation uses probabilistic distributions to characterize the work-load parameters. The simulation languages offer a rich choice of such probability distributions, which can be used by the work-load model to generate input data to the computer system simulation model according to the statistical data. For instance, an empirical distribution is usually used to generate instructions to be executed by the system model when the percentage of each instruction in the work load is known. When the simulation model is compiled, the simulation language generates input events according to the programmed probability distribution. A statistical simulation model is usually a good trade-off between the model simplicity and the accuracy of the simulation results.

When traces or statistics are not available, as in the case of modeling a completely new computer architecture, full computational emulation rather than control flow simulation can be done. In PAWS, for instance, User nodes and/or Compute nodes can be used to simulate the instruction execution. The registers of the architecture and the accessed memory locations can be simulated by using global variables. This model allows performance modeling and full modeling of the new architecture, using as input the code produced by a compiler. The work-load trace and statistics (percentage of each executed instruction, percentage of each addressing mode, taken and not-taken branches) can also be obtained as a result of this type of simulation. The cost is higher model complexity and, as a result, longer model turnaround and simulation runtime.

Building a simulation model in a very high level language takes advantage of the language's rich set of high-level modeling primitives. These primitives are used to simulate the contention between the system modules when executing each type of transaction (event) and the time spent by each transaction in each server.

The computer system model consists of a sequence of transactions executed by the model, active and passive nodes, a description of the way transactions circulate among the nodes, and runtime information. The transactions represent the unit of information processed by the simulated system. Both the information routing through the model and the information processing at every node are functions of the nature of the information (that is, functions of the transaction's type).

The simulation model nodes process each transaction as programmed. The simulation language should provide generic node types that satisfy virtually all modeling needs. The resource management nodes, for instance, represent the resources of the modeled system. Since transactions might have to wait for these resources, these nodes have queues associated with them.

The resource management nodes can be active or passive. The active nodes are servers, which provide service to a transaction such as the ALU. The passive nodes represent resources that do not perform work but must be possessed by a transaction to operate (such as buses). A passive resource is usually represented in a model by two nodes, one at which the resource is acquired and one at which the resource is released.

Each model uses simulation variables, handled automatically by the language, and user-defined variables. The user-defined variables can be global (accessible by all transactions) or local to the transaction for which they were created. The system model can also use submodels, which allow the top-down development of complex models. The submodel concept allows the programmer to build a model in a hierarchical fashion, by partitioning it into smaller pieces of the desired resolution. This structured approach greatly improves the model turnaround time and makes the model easy to understand and modify.

The simulation output should provide answers to all the questions of the computer architect and of the system designer. The requested performance measurements are usually those of throughput, resource utilization, average queue length and queue length distribution, and queueing time. Many times the system architect or designer needs a plot of one of these measurements as a function of one or more parameters. Ideally, the simulation language should provide a way for the programmer to request the plots needed; the plots should then be generated automatically at runtime.

Animation is another desirable feature of a performance simulation environment. An animation capability allows the computer system architect to display the movement of transactions in the model and the accompanying change in performance measures. This capability is important both as a teaching tool (showing the information flow inside the model for each type of transaction) and as a model debug tool.

Performance modeling example

We exemplify the use of a high-level simulation language by modeling in PAWS the performance of a hypothetical CPU when used in a hypothetical system environment. The simulated CPU (see Figure 1) does not represent a CPU from National Semiconductor or any other company. Its very limited (five instruction types and four addressing modes), simulated instruction set keeps the example simple. The external system built around this CPU consists of a memory management unit, an external cache, main memory, and DMA controllers. The purpose of the simulation is to understand the bottlenecks of the CPU internal architecture and to plot the system performance dependence on the cache hit ratio and the main memory access time.

The simulated CPU is pipelined; that is, several instructions are processed simultaneously in different execution stages inside the chip. An 8-byte-deep instruction queue, or IQ, supplies the instruction execution pipeline with prefetched instructions. The instruction prefetch request is triggered every time there are less than four bytes left in the IQ. The bus interface unit BIU responds to this request by prefetching and storing in the queue four new code bytes. At the same time the instruction decoder ID stage of the pipeline decodes instruction N of the executed program. This decoding is done by removing from the IQ the appropriate number of bytes, separating their information into opcode, addressing mode fields, and latching these fields so they can be used as inputs by the execution unit EXU and the address unit AU.

While instruction N is decoded, the AU prefetches the operands, if any, for instruction N-1 and stores them in the EXU operand queue. In parallel the EXU stage might execute instruction N-2, and results of previously executed instructions with destination memory might be waiting in a two-deep write FIFO (the pipeline gives priority to reads). Our model neglects the effects of pipeline interlocks (for example, trying to read data that was not written yet by a previous instruction) on performance. If necessary, their effect can be simulated using statistics or trace data.

For simplicity, the system simulation in this example is a statistical simulation, using statistical distributions for work-load modeling. Because the work-load model (the Code Generator CGN submodel) is separate from the system model, one can perform a deterministic simulation by simply replacing the current work-load model with a trace-driven work-load model.

The system model is implemented as a hierarchy of nested submodels (structured modeling approach). This method keeps the model simple and easy to understand, because it represents a block or a func-

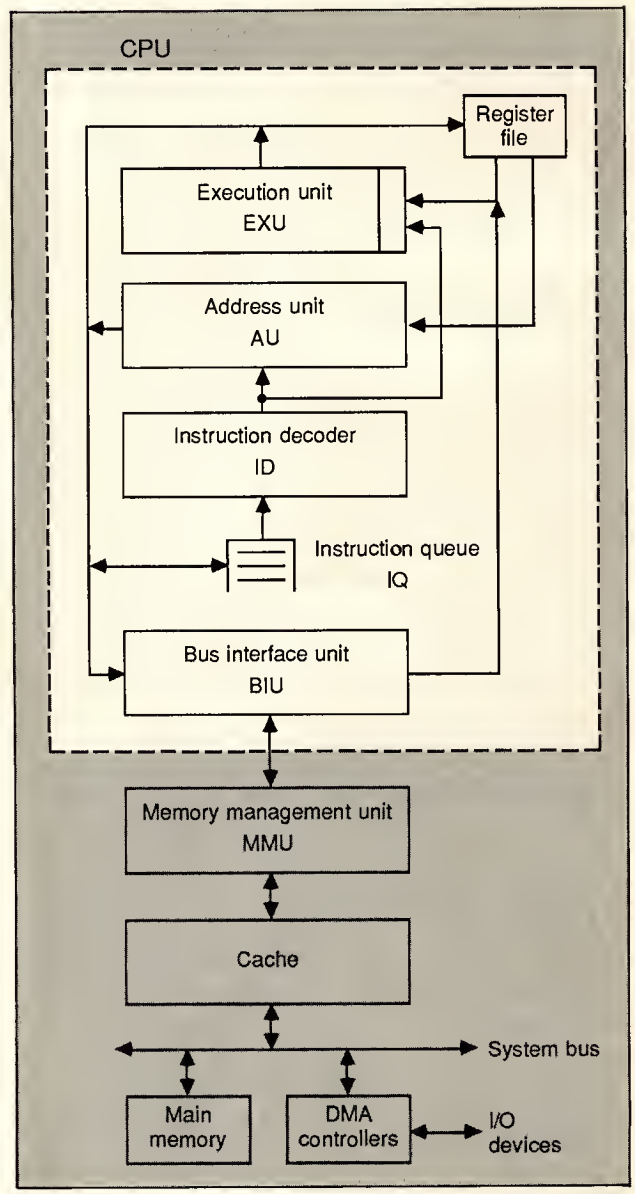


Figure 1. Simulated system using pipelined CPU. The CPU modules allow instruction prefetch, instruction decode, effective address calculation and operand prefetch, instruction execution, and write of results to memory, to be performed in parallel.

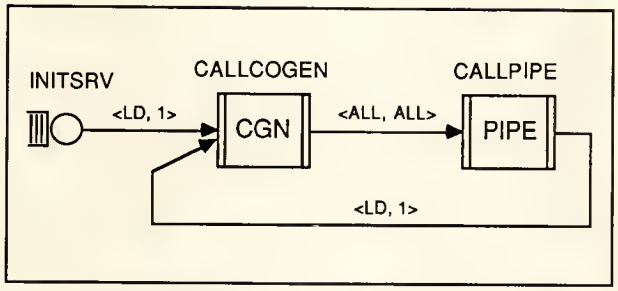


Figure 2. The main model is composed of three nodes: INITSRV to start the code generation, the CGN submodel for code generation, and the PIPE submodel for the simulated system. Special transaction <LD,1> indicates to the CGN that the pipeline is ready to accept a new instruction. The CGN answers by sending the next instruction to be executed to the pipeline, together with the local parameter values associated with this instruction.

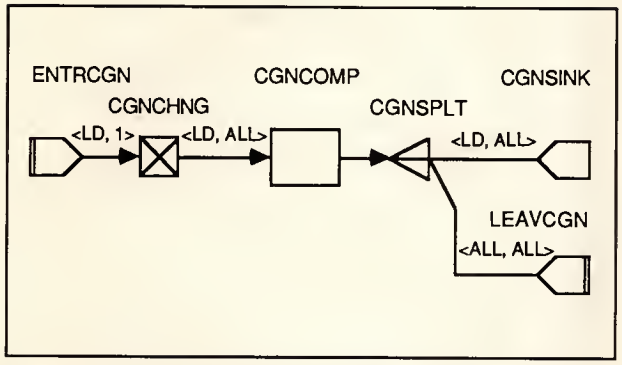


Figure 3. The CGN submodel generates new transactions according to the instruction mix and initializes a transaction's local variables according to the instruction type.

tion as a submodel without including all the implementation details for that block or function. Whenever a system module is requested to perform work for other system modules, this module is represented by a submodel called by the requesting modules. Different higher level submodels of the system can call the same lower level submodel. In this case the nature of the service requested by the higher level submodels is indicated by the type of the transaction entering the lower level submodel.

In our model, for instance, the IQ submodel is called by the ID, the AU, and the EXU submodels. Each of these higher level submodels generates special transactions to indicate to the IQ submodel the nature of the requested service (opcode bytes from the IQ for the ID submodel, displacement or immediate operand bytes from the IQ for the AU submodel, or IQ flush for the EXU submodel).

At its highest level, the simulated system simply consists of the main model displayed in Figure 2. This model consists of a CGN submodel, which generates the instruction transactions and the CPU instruction execution pipeline submodel, PIPE, which executes the instructions.

The CGN in Figure 3 is the work-load model. This submodel generates the transactions that represent the executed instructions. These transactions are generated in our model according to the available statistics (percentage of each type of instruction). The local parameter values of the instruction transactions are set by the CGN according to the instruction type and the addressing mode distribution (percentage of each addressing mode type).

The PIPE submodel seen in Figure 4 simulates the CPU instruction execution pipeline, consisting of the

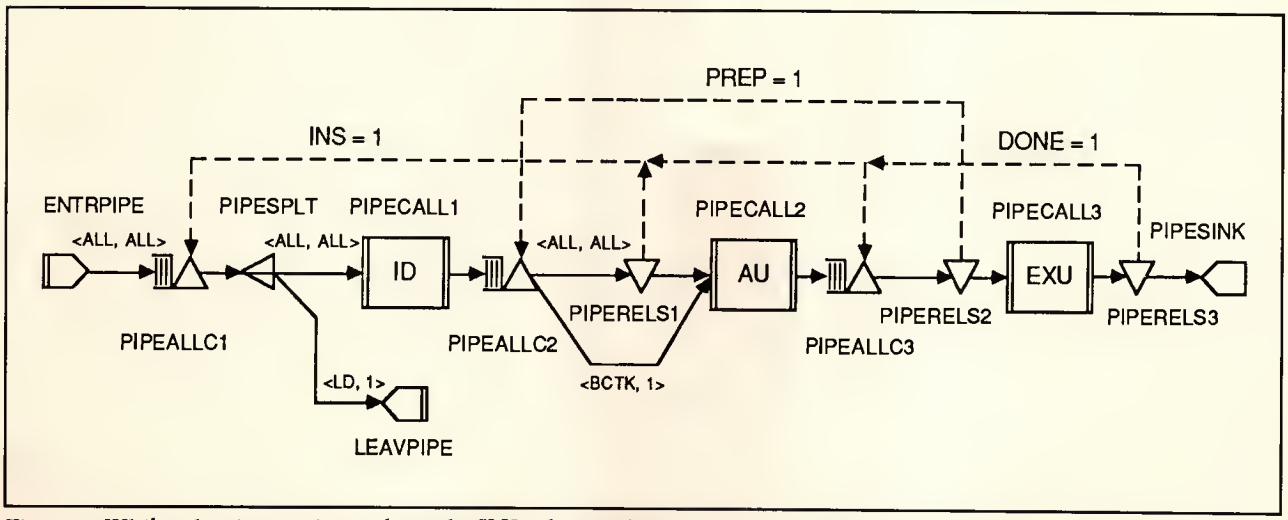


Figure 4. While other instructions release the INS token at the PIPERELS1 node to allow the pipeline to be refilled with instructions, the branch-taken instruction <BCTK,1> bypasses PIPERELS1 and releases the INS token at PIPERELS3, after exiting the EXU submodel. By doing this, we simulate in a simple way the behavior of the pipeline in case of a taken branch, without complicating the model with a pipeline flushing mechanism.

ID, the AU, and the EXU. Pairs of Allocate-Release nodes simulate the pipeline interlocks, while tokens associated with these nodes indicate whether or not the pipeline stages are ready to accept new instructions. When the pipeline is ready to accept the next instruction transaction, it signals its intent to the CGN by spawning a special transaction <LD,1>.

Whenever a taken-branch instruction is recognized by the EXU of our hypothetical CPU, the previous pipeline stages (as well as the IQ) are flushed to continue the instruction execution with the branch target instruction. We simulate the pipeline behavior in case of a taken branch by not allowing subsequent instructions to enter the AU and EXU submodels until the "branch-taken" transaction exits the EXU submodel (which implies that the IQ is flushed already). This step avoids complicating the model with a pipeline flushing mechanism.

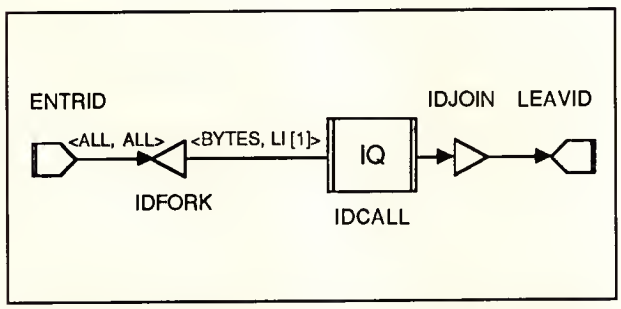


Figure 5. The instruction transaction entering the instruction decoder (ID) submodel waits at Fork node IDFORK for opcode bytes from the IQ and generates a new transaction of category <BYTES>. <BYTES> proceeds to the IQ submodel and then to the IDJOIN Join node.

The ID submodel detailed in Figure 5 simulates the instruction decoder stage. This stage simply requests the necessary number of opcode bytes from the IQ and prepares the proper control information for the next pipeline stages. The ID submodel simulates this stage by generating a new transaction of category <BYTES>, whose phase indicates to the IQ submodel the number of necessary bytes from the queue. After the IQ submodel finishes servicing this request, the original instruction transaction continues.

The AU submodel (Figure 6) simulates the effective address calculation of the operands, and the operand fetches (memory or register) started by this unit (the fetched operands are latched as inputs to the EXU). The CPU instructions might have, depending on their type, zero, one, and two operands. They all spend a minimum of two clocks in the AU.

The instruction transactions entering the AU submodel request the necessary number of operands for the EXU by generating the proper number of new transactions of category <GEN>. The original transaction waits for all the <GEN> transactions to be processed (equivalent with all the operands available to the EXU) before exiting the AU submodel.

For the case of immediate operands or effective address calculations for some addressing modes, the AU has to get immediate or displacement bytes from the IQ. The AU submodel simulates this by generating new transactions of category <BYTES> (none if no such bytes are needed). If some instruction operands are in memory, operand-fetch bus transactions are requested from the BIU submodel to fetch the operands to the EXU. These memory operand fetches are simulated in the AU submodel by generating new

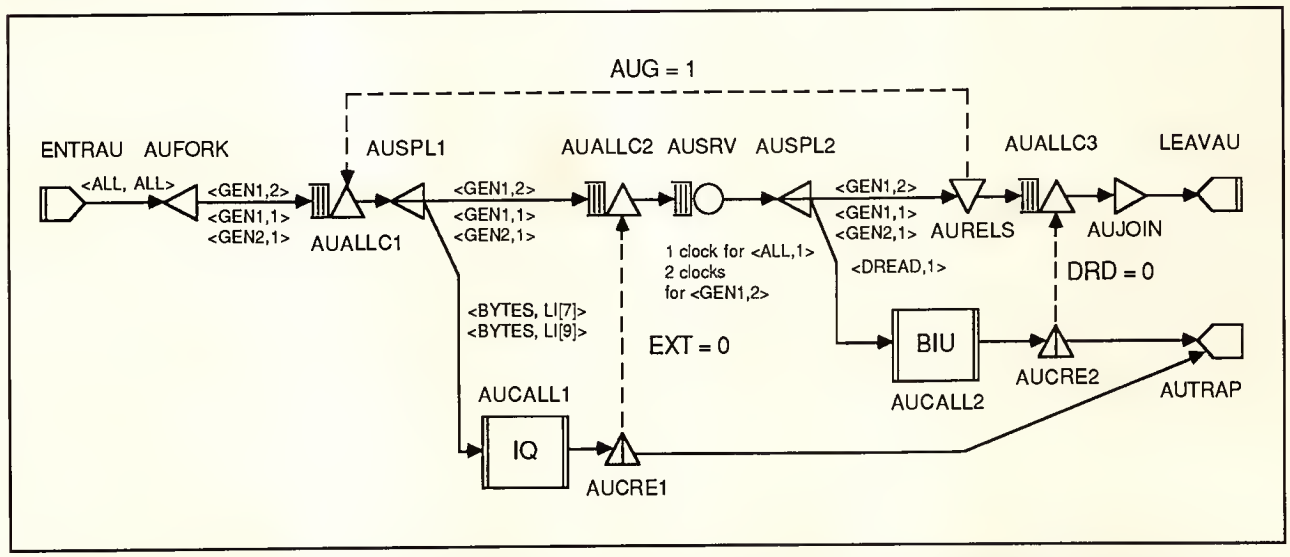


Figure 6. The operand reads executed by the address unit (AU) are simulated using special transactions <GEN1> and <GEN2>, which are queued at the AUALLC1 Allocate node waiting to be processed by the AU. In the hypothetical CPU we model, the AU spends a minimum of two clocks to service each instruction. This time is simulated in our model by the AUSRV Service node.

<DREAD> transactions. The <DREAD> transactions are interpreted by the BIU submodel as operand requests. Though the AU submodel description is relatively long and complex, its corresponding transaction flow graph in Figure 6 makes the submodel understanding much simpler. It also hints at how difficult it would be to build and maintain this system's model without the structured modeling made possible by the submodel concept.

The EXU submodel in Figure 7 simulates the CPU instruction execution. Each instruction execution time and the other actions initiated by the instruction are simulated. For instance, when a branch-taken instruction is recognized by the CPU EXU, the PC register is replaced with the branch target address, the pipeline is flushed, and the IQ is also flushed. As a result new code words are fetched from the branch target address.

We simulated the pipeline flushing effect in the case of a taken-branch transaction as part of the PIPE submodel. The EXU submodel simulates the IQ flush effect for this transaction by generating a new transaction of category <FLUSH>. This transaction is interpreted by the IQ submodel as an IQ flush request.

If the executed instruction has to write results back to the memory or read data from memory, these actions are simulated by generating new transactions. A <WRITE> transaction indicates to the BIU a memory write request; a <DREAD> transaction indicates a data read request. Our EXU submodel also simulates the existence of a two-deep write FIFO. This FIFO is used to buffer the memory write requests, allowing the EXU to proceed without waiting for the end of the write.

Several of the submodels described so far "called" the IQ submodel and the BIU submodel. The type of the requested service is indicated by the category and phase of the transactions entering these submodels.

The IQ submodel in Figure 8 on the next page simulates the CPU IQ. The transactions arriving at this submodel are either <FLUSH> transactions or <BYTES> transactions. The <FLUSH> transactions represent requests for IQ flush, while the <BYTES> transactions represent requests for opcode, immediate or displacement byte extraction from the IQ. (The number of bytes to be extracted is indicated by the <BYTES> transaction phase.)

A <FLUSH> transaction entering the IQ submodel destroys the tokens representing instruction bytes previously prefetched into the IQ before returning to the calling submodel. It also generates a new transaction of category <IREAD>, which signals the BIU to fetch extra instruction bytes. A <BYTES> transaction entering the IQ subtracts the number of requested opcode, immediate or displacement bytes from the token number representing instruction bytes previously prefetched into the IQ. The transaction also spends in the submodel the one clock cycle required by the IQ to return the requested immediate or displacement bytes and generates a new <IREAD> transaction before returning to the calling submodel.

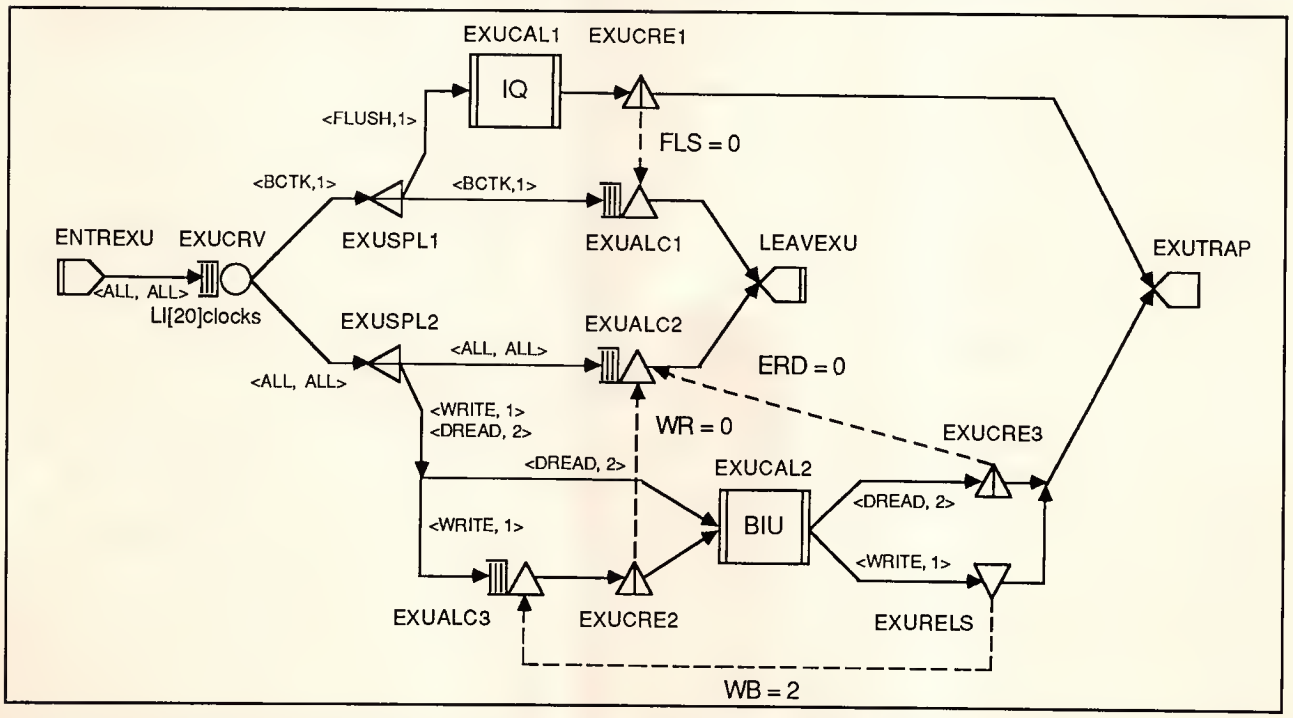


Figure 7. The EXUSRV Service node in the execution unit (EXU) submodel simulates the execution time of the executed instruction (in number of clock cycles). The execution time is indicated by a local variable of the transaction.

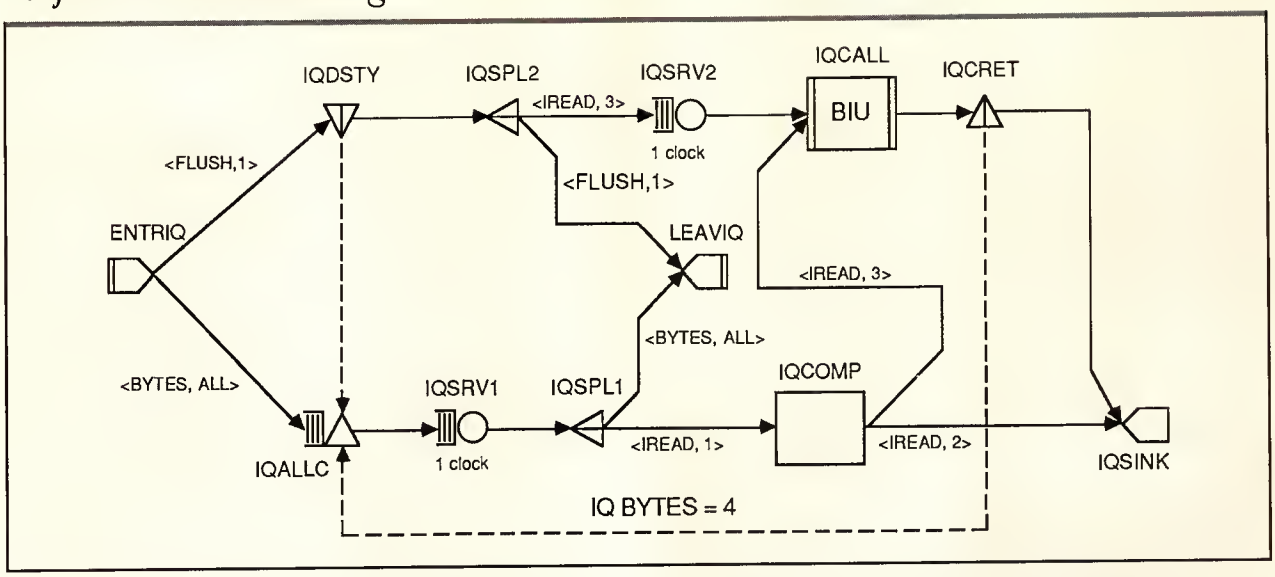


Figure 8. The transactions arriving at the instruction queue (IQ) submodel are either <FLUSH> transactions or <BYTES> transactions. <FLUSH> transactions move to a Destroy node IQDSTY that simulates the IQ flush. <BYTES> transactions are queued at the IQALLC Allocate node before proceeding to the IQSRV1 Service node. Here, they spend one clock cycle in simulation of the time spent by the ID or the AU in extracting bytes from the IQ.

The <IREAD> transaction proceeds to a Compute node, which checks the number of instruction bytes left in the IQ. If the number of remaining bytes is smaller than or equal to four (the condition, in our hypothetical CPU, for requesting the prefetch of four extra instruction bytes), the Compute node changes the phase of the <IREAD> transaction to indicate this fact (the phase is changed from 1 to 3 in our model). The <IREAD> transaction with the new phase is routed to the BIU submodel. If the number of instruction bytes left in the IQ is larger than four (no need for prefetch), the <IREAD> transaction proceeds unchanged and is destroyed.

The BIU submodel in Figure 9 simulates the CPU bus interface unit and the external system consisting of a memory management unit, cache, and main memory. This submodel is called by other CPU submodels (AU, EXU, IQ) to request an access to the external system. These requests are serviced serially (no new bus cycle can start until the active one is finished). The submodel gives priority to data reads over data writes and to instruction fetches over data reads. (As previously mentioned, this is not necessarily what happens in the CPUs from National or other vendors.)

A transaction entering the BIU submodel spends one clock cycle at a Service node, which simulates the time spent by the transaction in the hypothetical MMU. The efficiency of the MMU translation buffer TLB is simulated statistically. A percentage equal with the TLB hit ratio (TLBHIT) of the transactions proceeds directly to the system (simulated by the CACHE submodel). The rest of the transactions (whose percentage is TLBMISS = 1—TLBHIT) first generate new transactions of category <TLB>, to simulate the MMU page table accesses, and proceed to the system only after these transactions are serviced. The percentage of the transactions that proceed each of the two ways is specified as part of the BIU submodel topology.

The CACHE submodel in Figure 10 simulates the memory hierarchy of our system. The external cache simulated in this system is a write-through cache (all the writes update not only the cache, if hit, but also the main memory). The MMU accesses to the page tables are also noncacheable. A percentage of the CPU reads equal to the cache hit ratio, CACHEHIT, is satisfied by the cache without the need to go to the memory. The other CPU reads miss in the cache (their percentage is CACHEMISS = 1—CACHEHIT) and go to the memory to fetch the missing data to the CPU and the cache. The cache behavior is simulated by routing a percentage of the CPU reads (<IREAD> and <DREAD> transactions) equal to CACHEHIT to a Service node (CACSRV), which simulates the access time of the cache (two clocks for our model). The CACHEHIT parameter is specified as a global variable, to allow us to change it easily when plotting the system throughput as a function of the cache hit ratio and main memory access time. All the other transactions entering the Cache submodel are routed to the MEM submodel, which simulates those system modules sharing the system bus.

The MEM submodel in Figure 11 simulates the main memory, the DMA controller(s), and the system bus (see Figure 1). To keep the example simple, we sized the cache line so that it can be filled with one memory transaction only (no burst). A Source node is used to simulate the DMA transactions of the work load. These transactions are generated by the Source node according to the chosen statistical distribution (*uniform* distribution for our model). The DMA transactions compete for the system bus, according to the chosen bus arbitration discipline, with the CPU transactions entering the MEM submodel.

In our system the DMA transactions have a higher

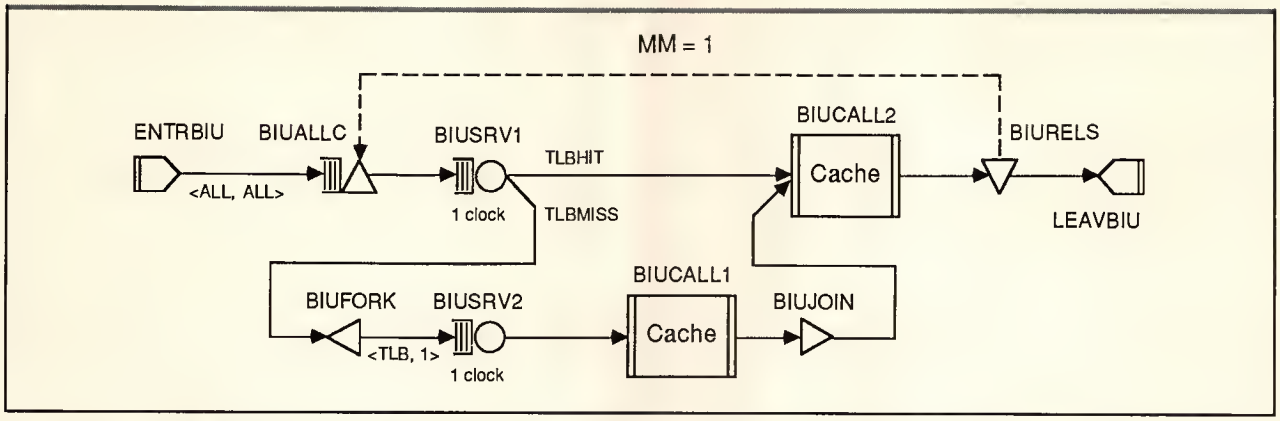


Figure 9. All the transactions entering the BIU submodel are queued at the BIUALLC Allocate node, waiting to acquire the MM token, which indicates that the CPU bus is available. The BIUSRV1 Service node simulates the one clock spent by the transaction in the MMU.

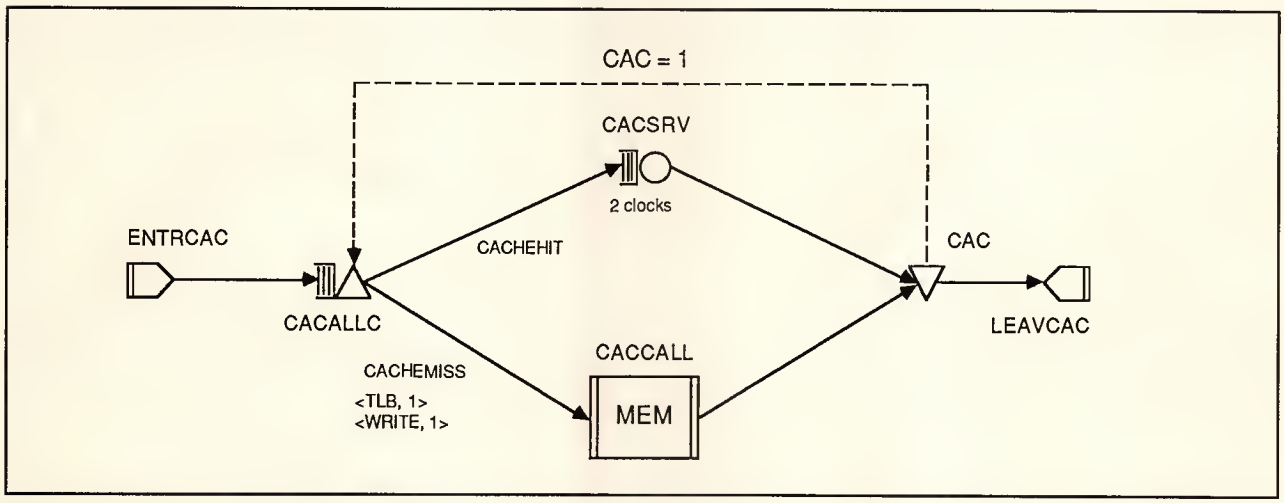


Figure 10. The external cache simulated in this system is a write-through cache. The MMU accesses to the page tables are noncacheable.

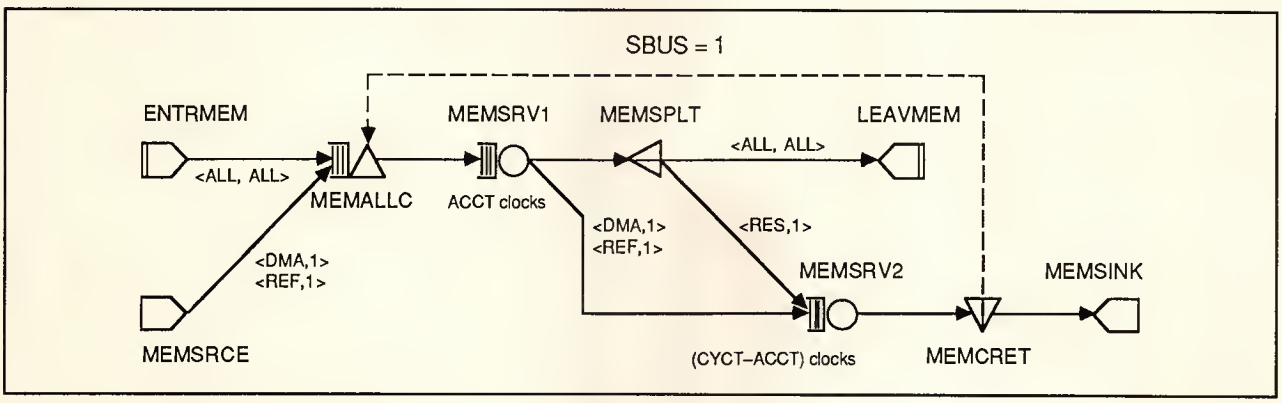


Figure 11. The main memory and I/O. The transaction with the highest priority queued at the MEMALLC node acquires the SBUS token, when available, and proceeds to the MEMSRV1 Service node. Here, it spends a number of clocks equal to the memory access time.

priority than the CPU transactions. The main memory access time is simulated by a Service node (MEMSRV1). The transaction that won the system bus spends at this node a number of clocks equal to the memory access time. (This access time is specified in our model as a global variable, ACCT, as our purpose is to plot the system throughput as a function of this parameter and the CACHEHIT parameter de-

*** BATCH NUMBER: 5 BATCH DURATION = 30000.000 (FROM 0. TO 30000.000) ***

(a)

		THROUGHPUT RATE	THROUGHPUT COUNT	QUEUE LENGTH MIN	QUEUE LENGTH MEAN	QUEUE LENGTH MAX	QUEUE LENGTH END	QUEUEING TIME MIN	QUEUEING TIME MEAN	QUEUEING TIME MAX	SERVICE MEAN	SERVICE UTIL
NODE: PIPE	/PIPECAL1 (1)											
CAT	/ALU1	0.005	164.									
CAT	/ALU2	0.009	256.									
CAT	/ALU3	0.010	293.									
CAT	/BCNT1	0.010	286.									
CAT	/BCTK1	0.006	171.									
CAT	/CMP1	0.002	71.									
CAT	/CMP2	0.004	112.									
CAT	/CMP3	0.004	113.									
CAT	/MOV1	0.010	288.									
CAT	/MOV2	0.016	491.									
CAT	/MOV3	0.021	622.									
CAT	/ALL	0.096	2867.									
NODE: PIPE	/PIPEALC2 (1)											
CAT	/ALU1	0.005	164.	0	0.049	1	0	0.	8.991	31.647		
CAT	/ALU2	0.009	256.	0	0.075	1	0	0.	8.809	33.000		
CAT	/ALU3	0.010	292.	0	0.086	1	1	0.	8.826	32.952		
CAT	/BCNT1	0.010	286.	0	0.088	1	0	0.	9.242	37.000		
CAT	/BCTK1	0.006	171.	0	0.054	1	0	0.	9.405	38.000		
CAT	/CMP1	0.002	71.	0	0.023	1	0	0.	9.659	26.449		
CAT	/CMP2	0.004	112.	0	0.031	1	0	0.	8.223	29.000		
CAT	/CMP3	0.004	113.	0	0.034	1	0	0.	9.120	27.000		
CAT	/MOV1	0.010	288.	0	0.084	1	0	0.	8.730	33.000		
CAT	/MOV2	0.016	491.	0	0.139	1	0	0.	8.519	32.000		
CAT	/MOV3	0.021	622.	0	0.183	1	0	0.	8.821	40.000		
CAT	/ALL	0.096	2866.	0	0.846	1	1	0.	8.855	40.000		

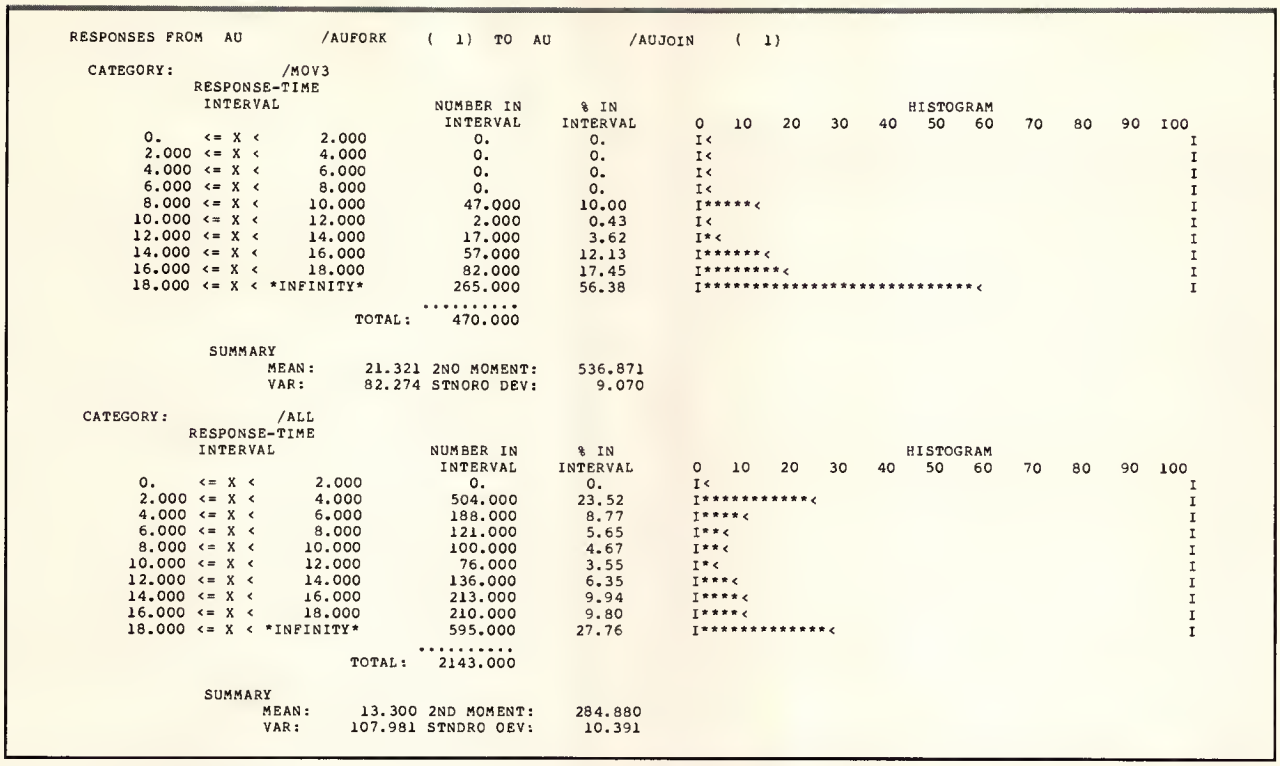
(b)

		THROUGHPUT RATE	THROUGHPUT COUNT	QUEUE LENGTH MIN	QUEUE LENGTH MEAN	QUEUE LENGTH MAX	QUEUE LENGTH END	QUEUEING TIME MIN	QUEUEING TIME MEAN	QUEUEING TIME MAX	SERVICE MEAN	SERVICE UTIL
NODE: PIPE	/PIPECAL1 (1)											
CAT	/ALU1	0.003	104.									
CAT	/ALU2	0.006	182.									
CAT	/ALU3	0.007	222.									
CAT	/BCNT1	0.006	187.									
CAT	/BCTK1	0.005	136.									
CAT	/CMP1	0.002	54.									
CAT	/CMP2	0.003	89.									
CAT	/CMP3	0.003	96.									
CAT	/MOV1	0.007	215.									
CAT	/MOV2	0.013	389.									
CAT	/MOV3	0.016	470.									
CAT	/ALL	0.071	2144.									
NODE: PIPE	/PIPEALC2 (1)											
CAT	/ALU1	0.003	104.	0	0.042	1	0	0.	11.993	48.000		
CAT	/ALU2	0.006	182.	0	0.083	1	0	0.	13.728	47.000		
CAT	/ALU3	0.007	222.	0	0.086	1	0	0.	11.589	64.000		
CAT	/BCNT1	0.006	187.	0	0.067	1	0	0.	10.808	56.000		
CAT	/BCTK1	0.005	136.	0	0.055	1	0	0.	12.050	45.000		
CAT	/CMP1	0.002	54.	0	0.022	1	0	0.	12.489	49.000		
CAT	/CMP2	0.003	89.	0	0.036	1	0	0.	12.302	52.000		
CAT	/CMP3	0.003	96.	0	0.033	1	0	0.	10.228	40.000		
CAT	/MOV1	0.007	215.	0	0.089	1	0	0.	12.480	46.000		
CAT	/MOV2	0.013	389.	0	0.159	1	0	0.	12.233	67.000		
CAT	/MOV3	0.016	470.	0	0.193	1	0	0.	12.341	57.000		
CAT	/ALL	0.071	2144.	0	0.866	1	0	0.	12.114	67.000		

(c)

				RESPONSE TIME MIN	RESPONSE TIME MEAN	RESPONSE TIME MAX
FROM AU	/AUFORK (1)	TO AU	/AUJOIN (1)			
CAT:	/ALU1			2.000	2.000	2.000
CAT:	/ALU2			6.000	11.036	33.952
CAT:	/ALU3			9.000	15.921	44.000
CAT:	/BCNT1			4.000	4.014	6.000
CAT:	/BCTK1			3.000	3.018	6.000
CAT:	/CMP1			2.000	2.606	5.000
CAT:	/CMP2			6.000	10.454	28.000
CAT:	/CMP3			9.000	16.032	34.521
CAT:	/MOV1			2.000	2.590	6.000
CAT:	/MOV2			6.000	10.699	29.832
CAT:	/MOV3			9.000	16.118	41.000
CAT:	/ALL			2.000	9.999	44.000
FROM EXU	/EXUSRV (1)	TO EXU	/EXUALC2 (1)			
CAT:	/ALU1			2.000	2.000	2.000
CAT:	/ALU2			2.000	2.000	2.000
CAT:	/ALU3			2.000	2.000	2.000
CAT:	/BCNT1			2.000	2.000	2.000
CAT:	/BCTK1			2.000	2.000	2.000
CAT:	/CMP1			2.000	2.000	2.000
CAT:	/CMP2			2.000	2.000	2.000
CAT:	/CMP3			2.000	2.000	2.000
CAT:	/MOV1			2.000	2.000	2.000
CAT:	/MOV2			2.000	2.000	2.000
CAT:	/MOV3			2.000	2.000	2.001
CAT:	/ALL			2.000	2.000	2.001

Figure 12. Sample summary statistics: gathered during a run of the model in which the memory access time was four clocks and the cache hit ratio was 80 percent (a); gathered during a run of the model in which the memory access time was eight clocks and the cache hit ratio was 80 percent (b); and presenting the response time of two of the instruction execution pipeline main sublocks, AU and EXU, requested by the programmer in the Statistics section of the PAWS program (c).



```
RESPONSES FROM  AU        /AUFORK    (  1)  TO  AU        /AUJOIN    (  1)

 CATEGORY:           /MOV3
          RESPONSE-TIME
           INTERVAL                    NUMBER IN      % IN                              HISTOGRAM
                                        INTERVAL     INTERVAL     0   10   20   30   40   50   60   70   80   90  100
      0.     <= X <       2.000           0.           0.       I<                                                  I
  2.000      <= X <       4.000           0.           0.       I<                                                  I
  4.000      <= X <       6.000           0.           0.       I<                                                  I
  6.000      <= X <       8.000           0.           0.       I<                                                  I
  8.000      <= X <      10.000          47.000       10.00     I*****<                                             I
 10.000      <= X <      12.000           2.000        0.43     I<                                                  I
 12.000      <= X <      14.000          17.000        3.62     I*<                                                 I
 14.000      <= X <      16.000          57.000       12.13     I******<                                            I
 16.000      <= X <      18.000          82.000       17.45     I********<                                          I
 18.000      <= X < *INFINITY*          265.000       56.38     I*****************************<                   I
                                       ..........
                              TOTAL:     470.000

          SUMMARY
                MEAN:      21.321 2NO MOMENT:     536.871
                VAR:       82.274 STNORO DEV:       9.070

 CATEGORY:           /ALL
          RESPONSE-TIME
           INTERVAL                    NUMBER IN      % IN                              HISTOGRAM
                                        INTERVAL     INTERVAL     0   10   20   30   40   50   60   70   80   90  100
      0.     <= X <       2.000           0.           0.       I<                                                  I
  2.000      <= X <       4.000         504.000       23.52     I***********<                                       I
  4.000      <= X <       6.000         188.000        8.77     I****<                                              I
  6.000      <= X <       8.000         121.000        5.65     I**<                                                I
  8.000      <= X <      10.000         100.000        4.67     I**<                                                I
 10.000      <= X <      12.000          76.000        3.55     I*<                                                 I
 12.000      <= X <      14.000         136.000        6.35     I***<                                               I
 14.000      <= X <      16.000         213.000        9.94     I****<                                              I
 16.000      <= X <      18.000         210.000        9.80     I****<                                              I
 18.000      <= X < *INFINITY*          595.000       27.76     I*************<                                      I
                                       ..........
                              TOTAL:    2143.000

          SUMMARY
                MEAN:      13.300 2ND MOMENT:     284.880
                VAR:      107.981 STNDRO OEV:      10.391
```

Figure 13. Sample histogram of the AU response time. The histogram was generated by running our system model with a main memory access time of eight clocks and a cache hit ratio of 80 percent.

fined in the CACHE submodel.) The main memory in our system has a cycle time that is different from the access time. In simulating this, the CPU transactions entering the MEM submodel spawn a new transaction, 5RES,1[5], which holds the memory for a number of clocks equal to the difference between the memory cycle time and access time. For simplicity, we assumed in our model that this difference is always two clocks.

Some of the simulation results can be seen in Figure 12. These statistics are gathered by PAWS in the Summary Statistics file, which contains some of the most important simulation results (throughput, queue length and queue time, service time) in a concise, easy-to-read format. These simulation results are gathered and printed by PAWS for each type of transaction, as well as for all the transactions, at each node of each submodel of the system. Figures 12a and b show some statistics gathered in our model's Summary Statistics file for the PIPE submodel.

The throughput rate at node PIPECAL1 or node PIPEALLC2 (the highlighted lines) in these figures represent, for instance, the system throughput in instructions-per-clock rates. By multiplying this number with the operating frequency in MHz, the architect gets the system throughput in MIPS, or million instructions per second. For the simulated CPU at 20 MHz, for instance, the throughput is 1.92 MIPS when the main memory access time is four clocks and 1.44 MIPS when the main memory access time is eight clocks.

The response time statistics, requested by the programmer in the Statistics section of the PAWS program, and the token utilization statistics are also printed in the Summary Statistics. Figure 12c presents the response time of two of the instruction execution pipeline main sublocks: AU and EXU. These response time statistics clearly show that the AU is the performance bottleneck in our hypothetical CPU. The average time spent in the AU by an instruction is 9.999 clocks when the memory access time is four clocks and the cache hit ratio is 80 percent. The average response time of the ID and EXU blocks is much lower (two clocks or less). These numbers show the chip architect which module limits the system performance and hints at what should be improved.

In the case of the simulated CPU the performance is limited by the displacement extraction (we can see this by looking at the branch instruction execution time) and by the external memory (cache and main memory) access time. (The execution time of register-register instructions is much shorter than the execution time of memory-register instructions and memory-memory instructions.)

More detailed statistics gathered by PAWS during the simulation runs are stored in the Detailed Statistics file. Besides the statistics in the Summary Statistics file (but organized differently), the Detailed Statistics file might contain the histograms of the response time, queue lengths, and queueing times requested by the programmer in the Statistics section of the program. Figure 13 shows the histogram of the

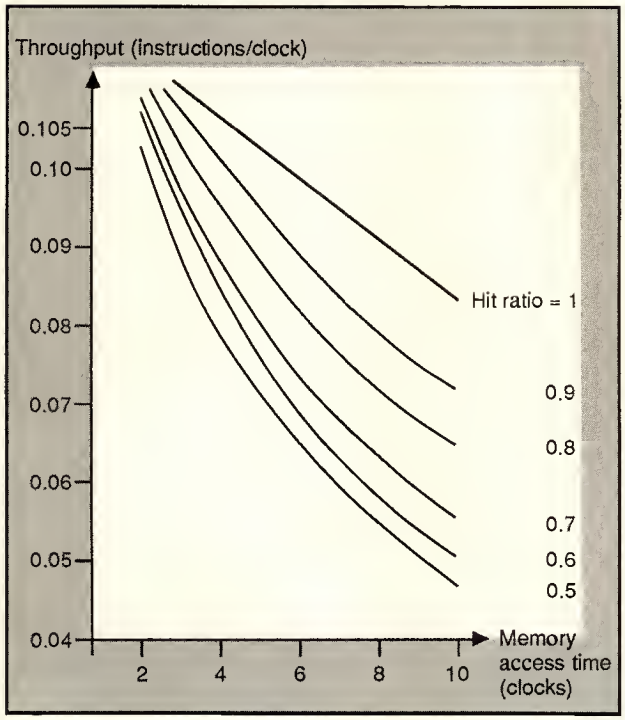


Figure 14. System throughput versus cache hit ratio and memory access time.

AU response time. The histogram was generated by running our system model with a main memory access time of eight clocks and a cache hit ratio of 80 percent.

Figure 14 plots our system's performance (throughput) as a function of the cache hit ratio and main memory access time. As previously mentioned, to generate this plot, we defined the cache hit ratio and the memory access time as global variables of the model and ran multiple simulations, each with a different value for one or both of these parameters.

Design decisions are very hard to make in today's complex VLSI chips and systems. Modeling the system helps the architects and designers evaluate its performance for a set of design assumptions when running the chosen work load. The system bottlenecks are also easy to determine when running such performance simulations.

New high-level simulation environments make system performance modeling fast and reliable by abandoning the procedural structure of the "classic" computer languages. These environments simply declare the elements of an input graph, which represents the information flow in the system. A rich set of very high level modeling primitives makes these simulation environments easy to learn and use by the system architect or designer. At the same time the VLSI chip and system models generated using these new simulation languages are easier to understand and maintain.

Acknowledgments

We would like to thank Max Baron, who guided and supported our performance simulation efforts, and David Schanin, whose encouragement and advice helped us to write this article.

References

1. J.F. Kurose et al., "A Graphics-Oriented Modeler's Workstation Environment for the RESearch Queueing Package (RESQ)," *Proc. 1986 Fall Joint Computer Conf.*, The Computer Society, Los Alamitos, Calif., pp. 719-728.
2. *Performance Analyst's Workbench System (PAWS 3.0), User's Manual,* Information Research Associates, Austin, Tex., 1987.
3. D. Ferrari, *Computer System Performance Evaluation,* Prentice-Hall, Englewood Cliffs, N. J., 1978.
4. B. Kumar, "Performance Evaluation of a Highly Concurrent Computer by Deterministic Simulation," MSCS thesis, University of Illinois, Urbana, Feb. 1976.
5. M. Baron and S. Iacobovici, "A Real-Time Performance Analyzer," *VLSI System Design,* May 1987, pp. 44-59.

Sorin Iacobovici is a senior computer and system architect with National Semiconductor in Santa Clara, California. His experience and interests include computer architecture, high-performance computer systems design, and performance analysis and modeling.

Iacobovici holds an MSEE degree from the Polytechnic Institute of Bucharest, Romania.

ChakChung Ng is a performance analysis specialist at National Semiconductor. His research interests and experience include computer system performance modeling and analysis, multiprocessor systems, cache systems, load balancing, and system-level simulation.

Ng is a PhD candidate at the University of California, Los Angeles, and a student member of the IEEE and the ACM.

Questions concerning this article can be addressed to the authors at National Semiconductor, M/S D3678, 2900 Semiconductor Drive, Santa Clara, CA 95051.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 159 Medium 160 Low 161

A HARDWARE SYNTACTIC ANALYSIS PROCESSOR

An innovative algorithm for syntactic analysis could be the first step toward placing grammar on a chip.

Mohammad Ali Sanamrad, IBM Nordic Laboratories
Koichi Wada, University of Tsukuba and
Haruya Matsumoto, Kobe University

A natural language processing system can have applications beyond the limited domain of its particular design, but only if it possesses a powerful and flexible processing unit completely separated from linguistic data. Such a processing unit can then serve as a natural language processor, and users will be able to incorporate it into their own natural language processing systems with appropriate linguistic data for various domains of a language and for different languages as well.

Most of the natural language processing systems currently in use are implemented using high-level programming languages such as Lisp or Prolog.[1-3] It has thus been necessary to build high-level programming language machines to speed up processing. However, Dubinsky and Sanamrad recently argued that certain levels of natural language processing, particularly that of syntactic processing, may not require the use of such high-level programming languages and could be performed using binary operations.[4,5] They then proposed an algorithm for syntactic analysis solely based on binary coding and processing of syntactic categories and relevant features. This innovation allows for microcomputer implementation of the syntactic analyzer as well as its hardware realization through available technology.

0272-1732/87/0800-0073$01.00 © 1987 IEEE

Here, as a step toward hardware natural language processors, we describe a hardware syntactic analysis processor design based on the algorithm for the binary syntactic analysis proposed by Dubinsky and Sanamrad. They proposed a set of categories, features, and phrase structure rules, and illustrated the workings of this analysis algorithm for a restricted coverage of English grammar. However, this does not mean that the categories, features, or rules should be fixed. The hardware syntactic processor presented here is designed so that the users are free to arbitrarily choose categories and features as required by the language and application area, and to design their own grammar rules based on these categories and features.

The parsing algorithm

We begin by describing the parsing algorithm as proposed by Dubinsky and Sanamrad.[4,5]

Category codes and feature codes. Each natural language word falls into one of the grammatical categories defined and coded in the system. Each grammatical category, or part of speech, is associated with a *feature code* into which the syntactic features relevant to that category are encoded. Grammatical categories such as "Noun" or "Det" appear in rules as *constituents*.

The syntactic features defined in the system may be either binary, which can be encoded using one bit, or *n*-ary, which requires the use of two or more bits. Thus, each bit in the feature codes specifies the status of a particular syntactic feature with respect to the constituent in question. For instance, the feature code corresponding to the pronominal *he* will have, among others, the bits representing "human," "singular," "masculine," and "third person" all set.

Phrase structure rules. The grammar is represented in the form of phrase structure rules.[6,7] Each rule has several left-hand constituents and one right-hand constituent, and is of the form

$$\text{Determiner Noun} \quad \begin{bmatrix} +\text{Count} \\ -\text{Determiner} \\ -\text{Deictic} \end{bmatrix} \rightarrow \text{Noun} \quad [+\text{Determiner}]$$

where square brackets contain the necessary feature specifications of the constituent in question, and ± indicates that the value for that specified feature should be 1 or 0, respectively.

Feature setting, checking, and transferring. Checking the feature requirements for the left-hand constituents of the rules is performed with the help of *feature selectors* and *feature evaluators*. These are a pair of code words, with the same length as that of the feature codes, that define feature specifications, if any, for each of the left-hand constituents. The feature selector indicates which features are to be looked at, while the feature evaluator indicates the requisite values of those features. The feature selectors and feature evaluators (as described) cannot determine that two or more bits are *not* set to a given value. It is therefore impossible to check the prohibitions required for the *n*-ary features; the use of *n*-ary features is not recommended except when necessary.

The application of each of the phrase structure rules involves the transfer of some of the feature code values from the left-hand constituents into the feature code of the right-hand constituent. A given rule may require the transfer of certain feature values from one constituent and other feature values from another constituent. To handle this transfer, each left-hand constituent is associated with a *feature code mask* which indicates which features are to be ignored and which features are to be carried over. Further, a rule may necessitate the setting of some feature values on the right-hand side which are not set in any of the left-hand constituents. For this purpose, each rule initializes a feature code (*initialized feature code*) for its right-hand constituent in which certain features may be preset to 1.

Dependency frame mechanism. In addition to the feature codes set for each constituent, a provision has been made for relating the main constituent of a phrase or sentence with each of its major dependents. For a restricted coverage of English grammar, for instance, the use of this mechanism can be restricted to verbs and predicates, although theoretically it can be utilized for any grammatical category. (For further information on grammatical terms used, see "Additional reading.")

The categories for which this mechanism is used, together with their encoded features, possess a *dependency frame number*. This number references a block of codes which specify features of the dependents for that particular category, as well as all others in its class. Dubinsky and Sanamrad have illustrated how this mechanism can work for verbs and predicates in English.[4,5] They have also argued that in languages which more fully encode agreement facts onto the verb (subject and object agreement for each of first, second, and third persons) such a unitary dependency frame system would require an enormous number of frames. It would be more efficient to use two separate frames:

- *a dependency frame* which would be common to all realizations of a particular verb type, and
- *an agreement frame* which would be common to any verbs bearing the identical set of agreement facts.

For example, while the verbs *has put* and *has run* would share a common agreement frame, the depen-

dency frame corresponding to *has run* would be the same as that of *have walked*. These two frames would then be merged for parsing.

The first code in the block of codes referenced by a dependency frame number is a *dependency generator,* which indicates the obligations or prohibitions of the dependents.

The dependency generator is followed by several pairs of codes that define any necessary feature specifications for the dependents in the form of feature selectors and feature evaluators described before. In this manner, one can specify that in English, for example, the verb *has put* minimally requires an animate subject, an object, and a locative adverbial.

If a rule which operates with reference to a dependency frame is applied, the constituents will first be evaluated against the dependency requirements and prohibitions of that dependency frame. If these conditions are met, the feature selectors and feature evaluators are copied out of the dependency frame into the feature selectors and feature evaluators associated with the related dependents in the rule, and parsing is continued as usual.

The parsing process. The parsing algorithm is illustrated in Figure 1. Note that the phrase structure rules are considered in order from the beginning and should be thus arranged in a way that the longest possible match is discovered first.

In the hardware later described in this article, categories, features, and rules are not fixed. The parsing algorithm is the only thing which is fixed and the processor is designed such that it is capable of manipulating arbitrary categories and features up to a reasonable maximum determined by the fixed length

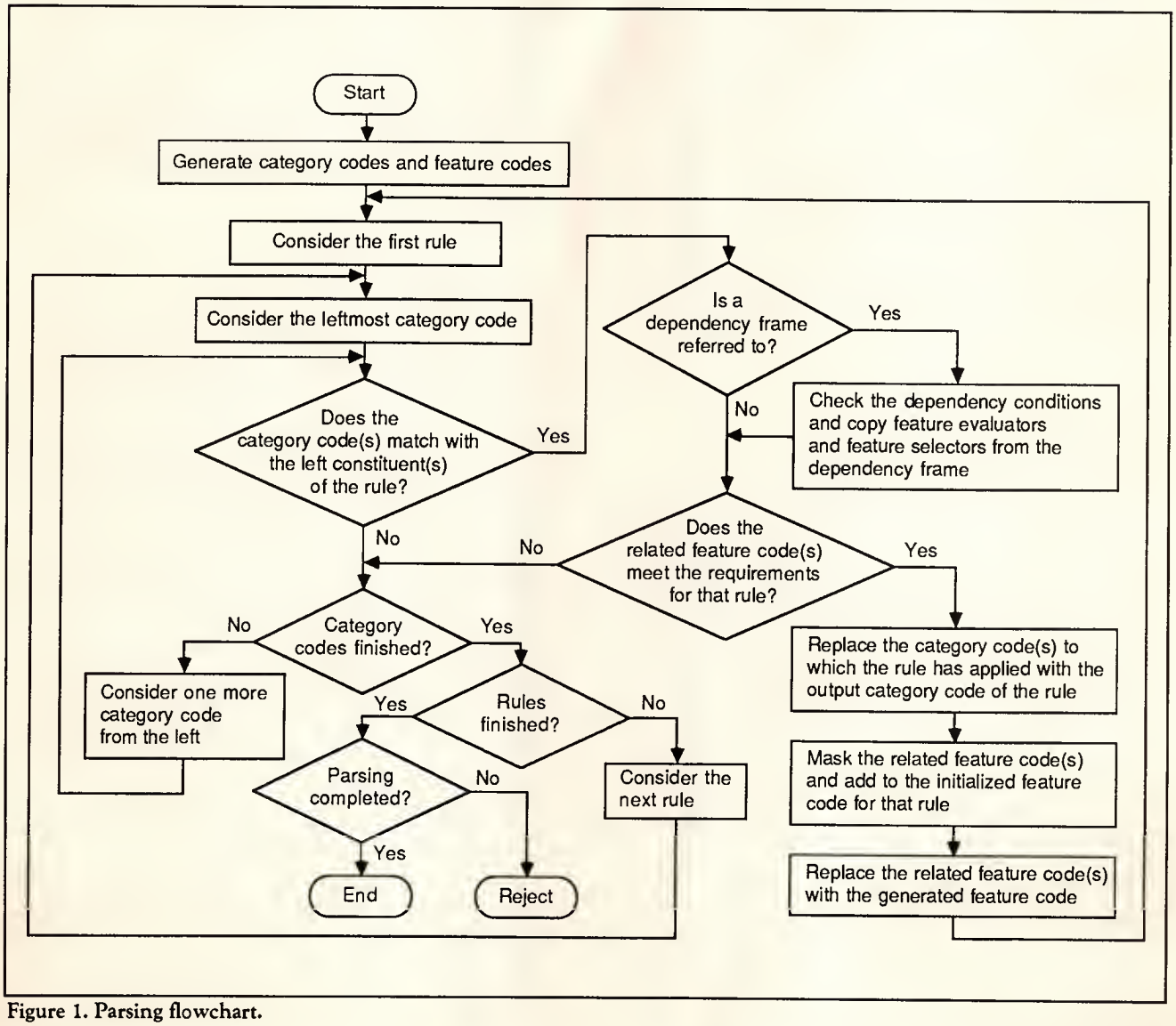


Figure 1. Parsing flowchart.

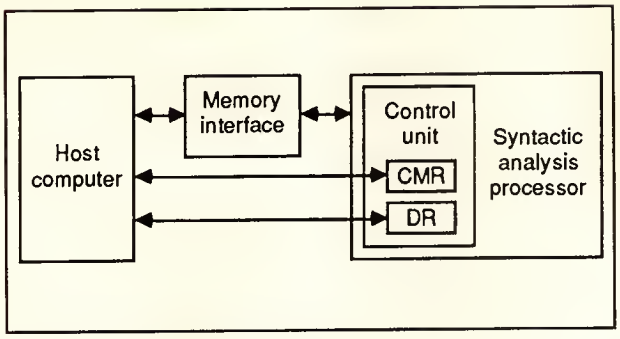


Figure 2. System configuration.

of the category code and feature code in the hardware.

Furthermore, while the above-mentioned mechanism for specifying features of the dependents is provided in the hardware, its use is not restricted to particular categories. This mechanism may be utilized for verbs and predicates in English, and for other categories in other languages. The dependency frames may be referenced either by a number directly taken from the related feature code, or by an address obtained through the merging of two or more frame numbers (such as a dependency frame number and an agreement frame number). Although the processor has been given all these abilities, none of them is fixed in the hardware for the sake of flexibility. Everything in this connection is controlled by the *control microprogram*.

System organization

The overall system configuration is shown in Figure 2. The syntactic analysis processor is connected to a host computer (such as a microcomputer) in the form of a back-end processor. I/O operations are then performed through the host computer.

The host computer is connected to the processor through a command register (CMR), a data register (DR), and a memory interface. The commands necessary for controlling the processor (that is, Run, Halt, Step Execution, etc.) are sent to the processor through the command register. The data register, on the other hand, is utilized for the data communication between the host computer and the processor.

Input sentences are first encoded into category codes and feature codes in the host computer. These strings of codes are then sent to the processor through the data register and the parsing starts as soon as a Run command is given by the host computer. The parsing results (that is, the resulting category code and feature code in the case of a successful parsing) are finally sent back to the host computer through the data register. Partial results, as derived at each step of the parsing, may also be sent to the host computer so that the process of parsing can be traced and, consequently, a parsing-tree can be built.

When the system starts, the host computer initializes and sets the internal memory and registers of the processor to the proper values. These settings are performed through the memory interface.

Hardware structure

The hardware structure of the processor is shown in Figure 3. The processor consists of an arithmetic and logic unit, a control unit, a matching unit, storages into which rules and other related codes are stored, storages where input sentences are encoded and stored for processing, and a number of registers for addressing these storages and communicating with the host computer.

The number of bits in category codes and feature codes has been fixed to 8 and 16, respectively. These lengths together with the size of the internal memory were chosen so that they would be sufficient for most practical purposes. The data communication inside the processor is performed through an internal 16-bit-wide bus.

Hardware components. The hardware consists of the following components:

- Arithmetic and logic unit, a bit-slice microprocessor which contains several registers and in which logical and arithmetic operations are performed.
- Control unit, containing a *writable control storage* (WCS) into which control microinstructions are written, and a bit-slice LSI sequencer (SEQ) which controls the execution of the microinstructions. This unit has also a pipeline register which provides a high-speed microinstruction fetch.
- Matching unit, a random-logic unit that checks whether the category codes of the input string currently addressed match with the left-hand constituents of the rule which is currently considered, and, at the same time, checks the related feature codes against the requirements for the rule as defined by the feature selectors and feature evaluators. If both of the matches are achieved, a *match flag* will be set. These matchings are performed in parallel to speed up the parsing process.
- *Left-constituent storage* (LCS), a 2K-byte memory which stores the category codes of the requisite left-hand constituents of each rule.
- *Right-constituent storage* (RCS), a 1K-byte memory which stores the right-hand category code of each rule.
- *Initialized feature storage* (IFS), a 1K-word (16-bit) memory which stores the feature code initialized for the output constituent of each rule.

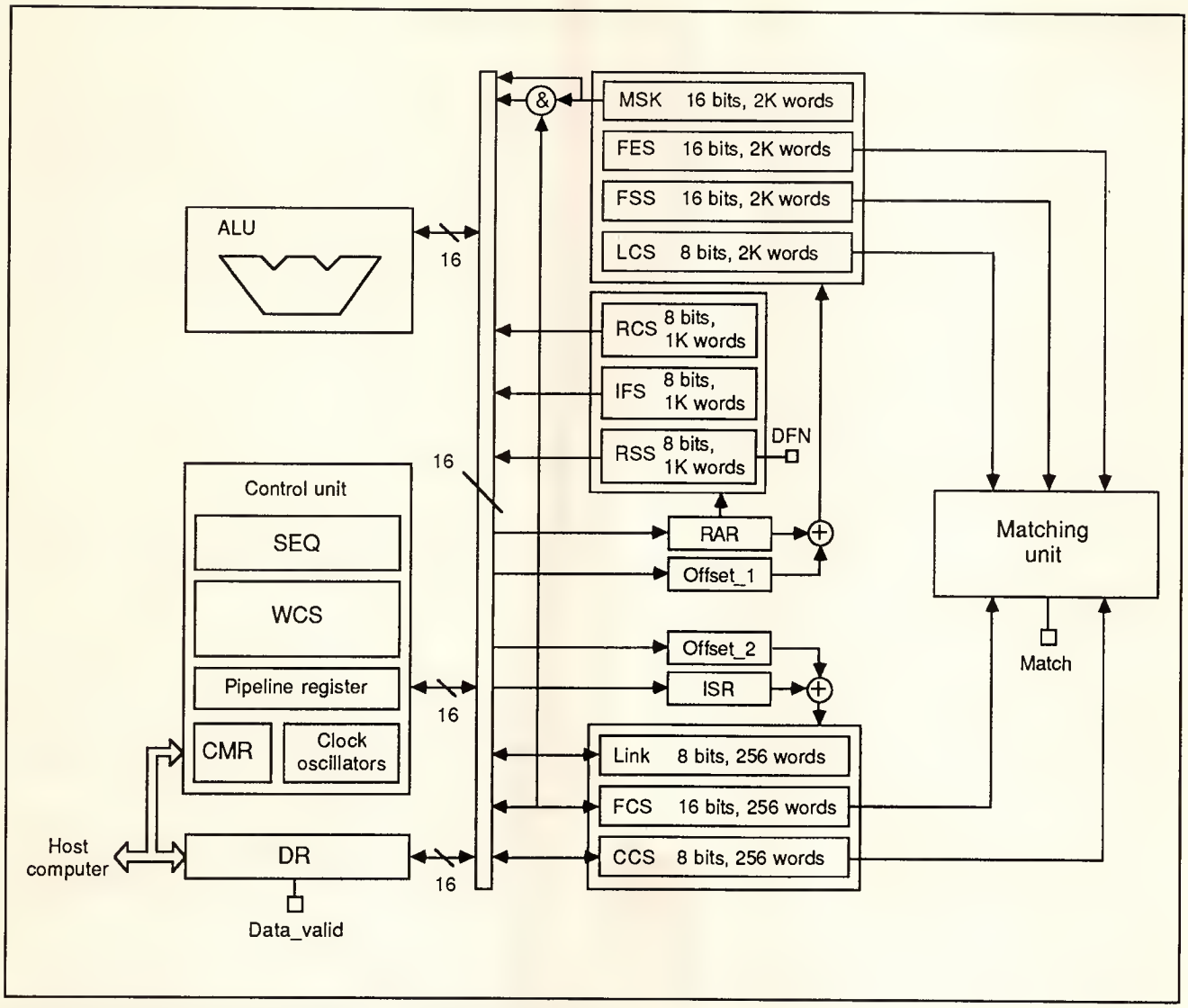


Figure 3. Hardware structure of the syntactic analysis processor.

- *Feature selector storage* (FSS), a 2K-word memory which stores the feature selectors for the left-hand constituents of each rule.
- *Feature evaluator storage* (FES), a 2K-word memory which stores the feature evaluators for the left-hand constituents of each rule.
- *Mask storage* (MSK), a 2K-word memory which stores the masks for the left-hand constituents of each rule. When the category codes and feature codes are being matched against the requirements of a rule, the feature code is automatically masked with the content of this storage and the result is ready at the bus to be taken by the ALU in the case of a successful matching.
- *Category code storage* (CCS), a 256-byte memory into which the category codes of the words of the input string are entered.
- *Feature code storage* (FCS), a 256-code word memory into which the feature codes of the words of the input string are entered.
- *Rule specifying storage* (RSS), a 1K-byte memory into which the number of the left-hand constituents of each rule together with other necessary specifications, if any, are stored. These specifications include whether or not a dependency frame is involved. In the case of referencing a dependency frame, one of the *dependency flags* will be set according to the associated category.

Internal-memory addressing. Rules are addressed by the *rule address register* (RAR). When a rule is addressed, the number of its left-hand constituents is read from the RSS and *offset_1* is incremented and automatically added to the content of RAR to pro-

duce the address of the codes associated with each of the left-hand constituents one after another. The same mechanism is also utilized for addressing the codes associated with the input string. The location of the category code and feature code, which are currently matched against the first left-hand constituent of the current rule, is given by the *input string register* (ISR). Again, to get the next codes to match against the other left-hand constituents of the same rule, *offset_2* is automatically added to the content of the ISR. The only difference here is that since the words in the input string are gradually merged as the parsing goes on, each coded word of the input string must indicate where the next word is in the memory. This indication is performed by storing the distance to the next word in the *link storage*. Thus, when addressing the next word, the content of this storage is loaded into offset_2. Figure 4 illustrates how this linking mechanism works in practice.

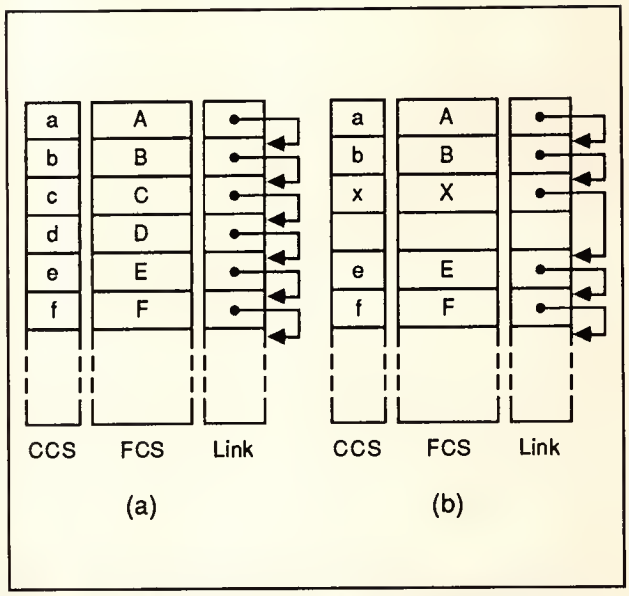


Figure 4. Linking mechanism: before (a) and after (b) applying the rule c d → x.

Dependency frames. Dependency frames are stored in the empty locations of the memory provided for storing the rules (MSK, FES, FSS, and LCS). When such frames are referenced, the specifications are read from this memory through the proper addressing of the control unit. The constituents of the input string are first evaluated against the dependency requirements and prohibition. Then the associated feature codes are checked against the feature requirements in the matching unit.

Control sequence

The control unit controls the operation of the ALU, addressing, and data communication through the internal bus. This is performed by the sequential execution of the microinstructions written in the WCS. The detailed operation of the processor can be followed in the control program written in Figure 5 in a Pascal-like language.

We have presented a hardware syntactic analysis processor based on the algorithm for binary syntactic analysis proposed by Dubinsky and Sanamrad. The processor is capable of parsing using a phrase structure grammar. Although it was originally designed for parsing in natural languages, as far as the categories and features are properly defined and rules are prepared, it is capable of parsing in other kinds of languages as well.

At each step of parsing, feasibility is confirmed by checking the requisite values of the features for the constituents to which a phrase structure rule is to be applied. When a rule is applied, a new feature code is set for the output of the rule by transferring some features from the rule input constituents and adding them into a feature code initialized for that rule in advance. The hardware is designed such that the rules, together with the codes necessary for setting, checking, and transferring of the features, can be adjusted to the application.

The processor can manipulate arbitrary grammatical categories and features designed by the user. The only restriction on the number of categories and features is due to the fixed length of the related code words and the internal bus in the hardware, which are fixed to a reasonable maximum in advance.

There is principally no restriction on the number of rules except for the memory size provided in the hardware. Of course, as the number of rules increases, the parsing time increases too. However, since everything is performed by the hardware, and because of the special mechanisms for addressing and matching provided in the hardware, there is no doubt that this processor would operate much faster than anything developed in a high-level programming language and running on a general-purpose processor.

Among other abilities provided in the processor, there is a dependency frame mechanism for relating the main constituent of a phrase or sentence with each of its major dependents. The use of this mechanism is not restricted to particular categories and is not fixed in the hardware. The hardware is rather designed such that it enables us to utilize the abilities of the processor, as needed by the language and application area, by writing appropriate control microinstructions in the processor. Thus, we may have lost a little bit of efficiency while buying a great deal of generality and flexibility.

```
 1: program parser;
 2: const   MASK = register number in ALU register file;
 3:         EOS  = End Of Sentence;
 4:         EOR  = End Of Rules;
 5:
 6: type Rule_Spec = record noc : 0..15;                    { number of categories }
 7:                         dfi : 0..15;                    { dependency frame indicator }
 8:                  end
 9:
10: var     { Rule Address Register, offset register_1,
11:           Input String Register, offset register_2 }
12:         rar,offset_1,isr,offset_2  : integer;
13:
14:         { Mask, Feature Evaluator Storage, Feature Selector Storage,
15:           Initialized Feature Storage, Feature Code Storage }
16:         msk,fes,fss,fcs            : array[0..2047] of word;
17:
18:         { Left Constituent Storage, Link,
19:           Category Code Storage }
20:         lcs,link,ccs               : array[0..2047] of byte;
21:
22:         { Initialized Feature Storage }
23:         ifs                        : array[0..1023] of word;
24:
25:         { Right Constituent Storage }
26:         rcs                        : array[0..1023] of byte;
27:
28:         { Rule Specification Storage }
29:         rss                        : array[0..1023] of Rule_Spec;
30:
31:         { ALU register }
32:         reg                        : array[0..15] of word;
33:
34:         { flag }
35:         match                      : boolean;
36:
37:
38:
39: procedure kernel;
40: begin
41:   par := 0;
42:   repeat
43:     offset_1 := 0;
44:     isr := 0;
45:     offset_2 := 0;
46:     repeat
47:       while match do
48:         begin
49:           if offset_1 < rss[rar].noc then
50:             begin
51:               offset_1 := offset_1 + 1;
52:               offset_2 := offset_2 + link[isr + offset_2];
53:             end
54:           else
55:             begin
56:               fcs[isr]  := reg[MASK] | ifs[rar];
57:               ccs[isr]  := rcs[rar];
58:               link[isr] := link[isr + offset_2] + offset_2;
59:               exit(kernel);
60:             end;
61:         end;
62:       isr := isr + link[isr + offset_2];
63:     until link[isr + offset_2] = EOS;                  { until End of Sentence }
64:     rar := rar + rss[rar].noc;
65:   until rss[rar].noc = EOR;                            { until End of Rules }
66:   if link[link[0]] = EOS then
67:     success
68:   else
69:     fail;
70: end;
71:
72:
73: begin
74:   while true do
75:     begin
76:       kernel;
77:     end;
78: end.
```

Figure 5. Control program written in Pascal-like language.

This processor, though capable only of syntactic analysis, is ideally suited for utilization as a module in natural language processing systems in various application areas. It will simplify the task of parsing and drastically shorten the necessary processing time. Finally, it is hoped that this work will be a step towards integrated circuits for natural language processing. Such very small and very fast processors would have many applications, especially in connection with speech processing for which integrated circuits are already available.

References

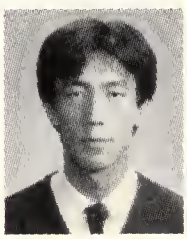
1. W.F. Clocksin and C.S. Mellish, *Programming in PROLOG*, Springer-Verlag, Berlin, 1984.
2. D.L. Waltz, "The State of the Art in the Natural Language Understanding," in *Strategies for Natural Language Processing*, W.G. Lehnert and M.H. Ringle, eds., Lawrence Erlbaum Associates, Hillsdale, N.J., 1982, pp. 3-32.
3. P.H. Winston and B.K.P. Horn, *LISP*, Addison-Wesley, Reading, Mass.,1981.
4. S. Dubinsky and M.A. Sanamrad, "Binary Coding of Universal Syntactic Features for a Natural Language Processor," *Preprints of 1984 Kansai Regional Joint Convention of Electrical Engineers of Japan*, Yamakatsu, Osaka, Japan, Nov. 1984, p. G270.
5. M.A. Sanamrad, "Machine Processing and Understanding of Spoken and Written Natural Languages," doctoral dissertation, Div. System Science, Grad. School Science and Technology, Kobe Univ., Kobe, Japan, Dec. 1984.
6. G. Gazdar, "Phrase Structure Grammar," in *The Nature of Syntactic Representation*, P. Jacobson and Geoffrey Pullam, eds., D. Reidel, Dordrecht, Holland, 1982, pp.131-186.
7. G. Gazdar, "Phrase Structure Grammars and Natural Language," *Proc. 8th Int'l Joint Conf. Artificial Intelligence*, Aug. 1983, pp. 556-565.

Additional reading

Leyons, J., *Introduction to Theoretical Linguistics*, Cambridge University Press, Cambridge and New York, 1979.

Mohammad Ali Sanamrad has been with IBM Nordic Laboratories in Lidingo, Sweden, since 1985. He received a BS in electrical engineering in 1978 from Arya-Mehr University of Technology, Tehran, Iran, and an MEng and PhD from Kobe University, Japan, in 1982 and 1985.

Sanamrad has authored several papers on processing and understanding spoken and written natural language. His current interests include natural language processing and logic programming. He is a member of the IEEE.



Koichi Wada is an assistant professor at the Institute of Information Sciences and Electronics, University of Tsukuba. His research interests include computer architecture for high-level languages, parallel computer architecture, artificial intelligence, and computer music.

Wada received a BE in electrical engineering in 1978, an ME in system engineering in 1981, and a PhD in computer science in 1984, all from Kobe University.

Haruya Matsumoto is a professor in the Department of Instrumentation Engineering at Kobe University. He has performed research in the fields of rocket and satellite observations of the upper atmosphere and electronic instrumentation. His current interests include the neural network and its application, electronic instrumentation, and space observation.

Matsumoto received his BE and his doctoral degree in electrical engineering from Kyoto University in 1953 and 1962, respectively. He is a member of the IEE of Japan, the Institute of Electronics, Information and Communication Engineers of Japan, and the Society of Instrument and Control Engineers of Japan.

Questions about this article may be directed to Sanamrad at IBM Nordic Labs, Box 962, S-181 09 Lidingo, Sweden.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

High 165 Medium 166 Low 167

MicroLaw

Richard H. Stern
Law Offices of Richard H. Stern
2101 L Street NW, Suite 800
Washington, DC 20037

Protecting hardware by copyright

Printed circuit boards and their audiovisual works

A perennial problem of hardware developers is how to keep competitors from appropriating their innovations. "Go get a patent" is a short and simple answer. But it is one that, most of the time, will not work. The problem is that a hardware innovation usually is too slight an advance over the prior work of others to justify a patent grant. Besides, patents cost $5000 or more before they are fully processed by the federal bureaucracy, and three years typically elapses during the process. In the interim, there is no remedy for infringement, and by the end of the process the market window for the product may have closed.

Copyright protection is ideal. The cost is minimal, and the protection attaches immediately. Interaction with federal bureaucrats is also minimal, in most cases. The fly in the ointment, however, is that—according to the standard mythology of copyright law—copyright protects only the expressions of ideas and not ideas themselves. A different expression of an idea will not, it is said, infringe the copyright on an earlier expression of that idea. Subject to a good deal of qualification, this rule usually applies.

Moreover, the utilitarian aspects of useful objects are ordinarily left unprotected by copyright law. Thus, a copyright in a drawing of a printed circuit board does not ordinarily protect the copyright owner from competing sales of the printed circuit board; the copyright just keeps competitors from making and selling the drawing itself. (The rule is otherwise in the United Kingdom and former Commonwealths. But elsewhere in Europe and in Japan, as in the United States, copyrights in drawings do not protect the objects depicted.)

A way around this ideological obstacle to fast, easy copyright protection for hardware products may now exist. Readers will recall that in the early days of video games there was considerable uncertainty about the strength of copyright in computer programs for video games. The proprietors of video games eventually got around that problem by asserting a copyright in the "audiovisual work" that their computer programs caused video game machines to perform. The courts were persuaded to hold that the collection of screen images (something like "sprites") generated by a video game program were copyrightable, and that their presentation on a screen in any order by a competitor was copyright infringement. In effect, the images were considered to be stored in EPROMs and the other electronic circuitry of the machine, which therefore comprised a medium (analogous to canvas for a painting or paper for a story) in which a copyrightable audiovisual work was "fixed."

This concept has recently been expanded to the Teddy Ruxpin mechanical teddy bears. Teddy Ruxpin has motor-actuated eyes, mouth, and other features. Information is stored in an ordinary audiotape cassette, in two channels. On one channel, there is simple audio storage, so that an amplifier and loudspeaker permit Teddy Ruxpin to "tell" a story. The other channel stores data that can activate the toy bear's internal servo motors, which results in his rolling his eyes, moving his jaws, and wrinkling his snout, while he tells his story.

Teddy Ruxpin, audiovisual toy.

When competitors tried to market their own stories on tape cassettes made to be compatible with the Teddy Ruxpin system, Teddy's proprietors sued for copyright infringement. And in two different federal courts, the judges agreed that the competitors were infringing a copyrighted audiovisual work, which included the collection of eye rollings, snout wrinklings, and other gestures performed by the bear when properly actuated. The analogy with video games and their audiovisual works was apparently compelling.

In a sense, the second channel of the tape stored a crude program for actuating the motors. But it probably seemed too implausible to try to assert a computer program copyright in that information. Furthermore, the competitive tapes probably had their own sequence

Continued on page 84

MicroStandards

Michael Smolin
Smolin & Associates
3428 Greer Road
Palo Alto, CA 94303

Generating standards in the IEEE

The Institute of Electrical and Electronics Engineers is recognized and accredited by the American National Standards Institute as a standards-writing organization. This means that the IEEE adheres to ANSI rules in the generation of standards, that the IEEE standards are eligible for adoption as National Standards, and that they may be presented by the United States National Committee to international standards organizations for adoption.

The IEEE has a formal sitting committee, the IEEE Standards Board, which is chartered by IEEE bylaws with responsibility for standards development in every IEEE society. The IEEE Standards Board meets four times a year to consider standards-related matters. A permanent staff at IEEE headquarters in New York City supports the board. Under the IEEE Standards Board in the hierarchy are several permanent committees. Of interest here are NESCOM and REVCOM.

NESCOM, the NEw Standards COMmittee, considers applications from sponsors (usually a technical committee) for the authorization of new standards projects. NESCOM makes recommendations to the IEEE Standards Board for action. Projects may range from revising an existing standard to creating a draft proposal for a standard in a new area. Once the Project Authorization Request (PAR) is approved, a project number is issued, and the working group can formally proceed to develop its standards proposal.

REVCOM, the standards REView COMmittee, considers the proposed standard that has been developed by a working group and recommended for adoption by its sponsor through a formal, written consensus ballot. Recommendations to the Standards Board by NESCOM and REVCOM are usually approved.

The IEEE consists of member societies, such as the Computer Society, the Instrumentation and Measurements Society, and the Communications Society. I use the Computer Society here as an example of the way the standards development process works.

Within the CS, technical work is handled in technical committees. The Technical Activities Board (TAB) generally oversees and administers to various Technical Committees (TCs). TC chairs are appointed by the Vice President for Technical Activities (and approved from above). This VP is also the chair of the TAB. These TCs, the repository of their society's technical expertise, sponsor the standards development activity for the IEEE. A society may have, as the Computer Society has, a Standards Coordinating Committee (SCC), which may act as a sponsor of last resort. Also cases exist in which special committees are empowered to become a sponsor for a particular project.

How it starts

Frequently, an individual or group feels that there is a need for a new standard and undertakes to find a sponsor for that activity. The initial contact may be through the IEEE Standards Office, a society officer, a volunteer member involved with standards, or a committee involved with standards or other projects in that area. The contact is passed to a responsive potential sponsor. Sometimes a sponsor recognizes a need for a standards project without outside prodding. The sponsor may then expeditiously submit a PAR to the IEEE Standards Board.

When an interested sponsor is found, a presentation or proposal is made to encourage adoption of the project. This is conducted by mail to, or in person at a meeting with, the critical individuals involved.

The sponsor may

- agree to sponsor the project, or
- decide to study the issue in a study group, or
- agree to sponsor the project but change some aspects of the proposal to suit the sponsor, or
- table the issue, or
- reject the proposal and decline to sponsor the project.

In the latter case, another TC may be recommended as a more suitable project sponsor.

When a TC agrees to sponsor a project, the sponsor submits a PAR to NESCOM. The PAR identifies the project by title, scope, purpose, and sponsor. The PAR specifies other standards bodies with whom formal coordination will be undertaken during the project and the group that will write the draft. The working group may be formed and start meeting and working while awaiting a response to the PAR (up to three months). Sometimes a working group is formed in advance of the PAR submittal, and that working group recommends the wording of the PAR to the sponsor. In some cases a standards subcommittee may carry out those responsibilities of the sponsor TC that have been delegated to it by the sponsor.

The working group

The sponsor appoints the working group chair, who then forms and organizes the working group. The working group has few restrictions placed on it. It must have regular, open meetings; must record and publish minutes; and generally must act in a fair, open, and aboveboard manner. The sponsor is responsible for the proper running of its working groups. The task of the working group is to write a draft for the standard project authorized by its PAR.

The sponsor is responsible for the actual content of the drafts, and so sponsor approval is required before drafts can move beyond the working group. Depending on what approval is sought, the approval may come from the standards subcommittee, the sponsor chair, the sponsor standards director, the sponsor's standards subcommittee, or a vote of the sponsor membership. For example, since the sponsor is responsible for technical content, the sponsor chair must approve all requests to publish drafts. (Also required is approval of disclaimers and so on from the Vice President of Standards.)

The working group has the task of creating a working document (draft) for its sponsor to consider recommending to the IEEE Standards Board for adoption as an IEEE Standard. The members of the working group are expected to work as individuals promoting the good of the profession. Members of the working group vote as individual professionals. The working group must coordinate its activity with those other standards-generating bodies having interests in the subject being standardized. These bodies may include domestic groups like X3, other societies of the IEEE, SAE, or even the DoD, as well as international groups like ISO, IEC, and JTC1. Part of that coordination is formally specified in the PAR.

In the process of developing a draft working document for a standard, members of a working group are encouraged to author articles about their project for publication in society publications (for example, *Computer*, *IEEE Micro*), although publication of the draft itself is deprecated. Concerns center on a fear that archival copies will reside in libraries and perhaps be conformed to as if they were copies of an approved standard. Members of the working groups are also encouraged to present papers on the activity.

The working group must be responsive to comments from outside the working group, actually soliciting such comments from the interested and concerned professional community. Articles and presentations encourage comments. Some of the comments will result in changes to the draft. All must be considered by the working group and should be responded to in writing.

Eventually, the working group considers that it has completed its task and votes the draft working document out of committee. It is then considered by the sponsor's standards subcommittee (if any).

The working group must be responsive to outside comments, actually soliciting them from concerned professionals.

The Computer Society's Technical Committee on Microprocessors and Microcomputers (TCMM) sponsors many standards projects. The TCMM has a standing subcommittee called the Microprocessor Standards Committee (MSC), which acts as the TCMM's standards subcommittee in overseeing the working groups sponsored by the TCMM. At the present, the TCMM is sponsoring 24 standards projects, all in various stages of development.

It is the function of the MSC to assure the sponsor that the draft is technically correct, that the document meets the goals for which the working group and project were sponsored, and that procedural requirements have been met. This standards subcommittee, the MSC, may reject the document, table consideration, direct the working group to make changes, make changes of its own to the document, approve the document, or take any other action that a committee normally would take regarding a document from one of its subcommittees. Action at the MSC usually takes the form of a motion passed by a show of hands at a meeting. Approval takes the form of recommending the draft to the sponsor.

Sponsor ballot

It is the responsibility of the sponsor TC to consider the recommended draft via formal mail ballot. This sponsor ballot should assure that the draft going to the Standards Board is the consensus of all interests significantly affected by the scope of the standard. The balloting may be delegated to a subgroup of the TC membership, plus others who represent concerned interests not otherwise represented.

Voters in the balloting body have 30 days to respond to the ballot (that is, get their vote in). This period is often extended to allow for the speed of the various "snail mail" services around the world.

The rules of the IEEE require that a vote be one of:

- approval, or
- approval with attached comments (often used to point out small or editorial errors),
- rejection—this *must* be accompanied by a detailed commentary designed to allow the changes to the draft that are necessary to turn the rejection into approval, or
- abstention, with reason.

For the ballot to be valid, 75 percent of the ballots must be returned as legitimate votes. For the proposal to pass, 75 percent of the nonabstaining returns must be favorable.

All negative comments received during the sponsor ballot must be considered by the sponsor, and responded to in writing. The received comments may result in the sponsor's making changes to the document. If significant or substantive changes are made, the members of the sponsor balloting body must be given a chance to consider the changes and an opportunity to change their votes. When all requirements are complied with and the vote is favorable, the sponsor recommends to the IEEE Standards Board (through REVCOM) that the proposed standard be adopted as an IEEE Standard.

The IEEE Standards Board adopts several classes of documents:

- guidelines,
- recommended practices,
- trial-use standards, and
- full standards.

More on these documents, and getting standards to international bodies, will appear in another issue. Meanwhile, here's an update on activities.

Current activity

At the June 1987 meeting of the IEEE Standards Board, these new projects for standards development were approved:

- P1151, Modula 2, a Modular High-Level Programming Language,
- P1152, Smalltalk, an Object-Oriented Programming Language and Environment,
- P1154, PILOT, A Program Instruction Learning or Teaching Language,
- P1155, A High-Speed Backplane Instrumentation Bus,
- P1156, Connectors and Mechanical Packaging for High-Reliability Bus Structures, and
- P1496, Rugged Bus, a Very High Reliability Bus Structure.

The Technical Committee on Microprocessors and Microcomputers sponsored these projects, and its oversight standards subcommittee, the Microprocessor Standards Committee, administered them.

The TCMM, again the sponsor, withdrew the P1153, Program Development Language Project Authorization Request in favor of the new activity in X3. In addition, the TCMM brought four projects to the IEEE Standards Board for adoption as IEEE standards. P1101, P1196, and P1296 were adopted as IEEE standards and are now referred to as:

- IEEE Std. 1101-1987, IEEE Standard for Mechanical Core Specifications for Microcomputers,
- IEEE Std. 1196-1987, IEEE Standard for Nubus, a Simple 32-Bit Backplane Bus, and
- IEEE Std. 1296-1987, IEEE Standard for a High-Performance 32-Bit Bus.

Prepublication versions (the final committee drafts) are available from the IEEE Standards Office for a moderate fee.

P896.1 was adopted subject to the proper completion of the response to negative votes received as part of the consensus-forming sponsor ballot. When procedures are completed (shortly), it will become IEEE Std. 896.1-1987, IEEE Standard for the Futurebus Backplane Bus.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 174 Medium 175 Low 176

MicroLaw

Continued from page 81

of signals, so that it would be doubtful that the two alleged computer programs were alike enough for infringement. The alleged audiovisual work was less implausible than the alleged computer program as a vehicle for asserting intellectual property rights. Another alternative source of intellectual property rights could be a copyright in the appearance of Teddy Ruxpin—a sculptural copyright. The problem here, however, was that the competitors were *not* selling look-alike bears or clone Teddy Ruxpins. They were selling tapes of different stories, designed to cooperate with a genuine Teddy Ruxpin bear and cause him to tell a different story (like putting a new phonograph record on the customer's existing Victrola, a classic patent dispute from the early part of this century). Apparently, the facts of the Teddy Ruxpin cases call for the audiovisual work copyright theory or nothing.

Now, imagine the following hypothetical, imaginary scenario. You develop a mechanical toy, with motors actuated by a microprocessor and computer program. The toy waves its motor-actuated arms, legs, wings, fins, tail, or whatever in accordance with the dictates of the program. The toy becomes a great commercial success and you attract competition; competitors start cloning similar products, with just enough variations on the appearance of the toy to defeat a sculptural copyright. What then? Simple. You sue them for infringing the copyright in your computer program, if they copy it.

But suppose they write their own program? Then you sue them for copying the "look and feel" of your computer program, as explained in earlier issues of *IEEE Micro* (see MicroLaw, Apr. 1986, pp. 64-65; Dec. 1986, p. 78; and June 1987, p. 82.)

But suppose you do not use a microprocessor device? Suppose you use NAND gates, flip-flops, and the like—all hardware? Your competitor copies your printed circuit board. But your printed circuit board is not protectable as such under copyright law. Then what?

What is an audiovisual work?

Section 101 of the Copyright Act defines "audiovisual works" as "works that consist of a series of related images which are intrinsically intended to be shown by use of machine or devices such as projectors, viewers, or electronic equipment, together with accompanying sounds, if any, regardless of the nature of the material objects, such as films or tapes, in which the works are embodied."

The audiovisual work might come to your rescue. You might assert that the printed circuit board caused a set of images to appear in three dimensions. The images are the gestures, tail waggings, wing flappings, and so on executed by the mechanical toy when the printed circuit board activates its motors. The audiovisual work is stored or "fixed" (a copyright law requirement) in the printed circuit board. Voila! Injunctions, damages, seizure and destruction of offending toys, and the whole arsenal of copyright remedies. It may be hard to believe, but you know what Barnum and Lincoln said about the gullibility level. And the two Teddy Ruxpin cases are a precedent for this approach.

Can the concept be expanded from mechanical toys to other electronic products? Probably, Chuckie Cheese. . . . Perhaps, CAD/CAM?. . . What else?

Reader comment is invited.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 171 Medium 172 Low 173

MicroReview

Richard Mateosian
2919 Forest Avenue
Berkeley, CA 94705
(415) 540-7745

Microprocessor interfacing

I ran into Rodnay Zaks the other day. Zaks is the founder of Sybex Computer Books and is one of the earliest and most successful of the popularizers of microprocessor technology. Our conversation quickly turned to computer books, and he told me the subject the buying public is looking for: microprocessor interfacing techniques.

Zaks has survived the ups and downs of the computer book business, so I certainly didn't take this lightly. Furthermore, *Microprocessor Interfacing Techniques* by Zaks and Austin Lesea is one of the classics in the field. Zaks told me that it still sells well, even though it is long out of date, and I know that it has had nothing but good reviews ever since it came out in 1978. (Once, I even heard Adam Osborne praise it.) According to Zaks, people are hungry for books of this sort, but there are none to be had.

That kind of marketing information makes fortunes for people, but in this case it came too late. Just a few days after my conversation with Zaks I received a review copy of a book of exactly the sort Zaks wanted. Surprisingly it's not just a hack job thrown together to some marketeer's specification. It's a masterpiece, six years in the making but still totally up to date, worthy of becoming the standard work in the field for the next five years or more.

Microprocessor-Based Design, A Comprehensive Guide to Effective Hardware Design, Michael Slater, Mayfield Publishing (Mountain View, Calif., 1987, 632 pp., $39.95)

I don't read a lot of 600-page books, and I don't read from cover to cover every book that I review. Nonetheless, over a period of about a month I read almost all of this book—simply because I found it so interesting. Parts of it were interesting to me because they are orderly expositions, pitched at my level, of subjects I've had to deal with but have never felt comfortable with. Other parts dealt with subjects that I know well and that I've written about myself. These, surprisingly, were even more interesting to me, because Slater has solved some difficult problems of exposition, avoided widely held misconceptions, and focused on what I consider to be the key issues. In short, I like the book very much, and I recommend it to anyone who wants or needs a survey of microprocessor-based hardware design.

This book divides the body of knowledge required for microprocessor-based design into nine major areas, then covers each in a chapter. These areas are microprocessors, buses, memory, bus peripherals, user interface, displays, communications, mass storage, and multiprocessor systems. All of this material is covered competently, but the most inspired chapters, the ones in which the author's intimate experience with the material comes through most clearly, are the ones dealing with the first four areas mentioned above.

One gap in Slater's coverage of this material is the deliberate exclusion of software considerations. Each of the areas mentioned above is discussed strictly from the hardware point of view. This honest approach is preferable to an incompetent or superficial coverage of software issues, but it does leave a gap.

Each chapter follows a pattern. First is an introduction—not the usual formula of "tell them what you're going to tell them" but a summary of the key issues and relevant history. Next comes a survey of the different options designers have to work with. For example, the microprocessor chapter includes sections on 8-bit processors, 16/32-bit processors, 4-bit and 8-bit single-chip microcomputers, and 16-bit single-chip microcomputers.

Each of these sections contains descriptions of the principal products, discussions of how they work, analyses of key design decisions and their implications, and bits of lore that "everybody" working with these products knows. The text is amply supplemented by figures. Many of the figures are original, while many others are taken from manufacturers' literature. Almost all of the figures are clear and readable and make the intended point in an easily understood way.

The next section of each chapter compares the options just described and outlines selection criteria that a designer would use in choosing one of them. This kind of material is difficult to write, and Slater has done a good job of sidestepping manufacturers' propaganda and focusing on the real issues of importance to designers.

Next in each chapter comes a design example. The design examples from all of the chapters present different aspects of a dedicated 68008-based controller. Tying the design examples together in this way allows a realistic example to be

ENTER THE

Annual Gordon Bell Awards

FOR PARALLEL PROCESSING SPEEDUP

Our sister magazine, *IEEE Software*, will judge annually two $1000 awards aimed at promoting better parallel-processing applications. One prize will be for best speedup on a general-purpose (multiapplication) multiple-instruction, multiple-data processor; the other will be for best speedup on a special-purpose MIMD processor.

C. Gordon Bell, the director of the National Science Foundation's Computer and Information Science and Engineering Directorate, will fund the awards for the next 10 years. Bell said he offered to sponsor the annual awards to underscore his commitment to parallel processing.

Entries for the 1987 awards must be received by Dec. 1. The winners will be announced in *IEEE Software*'s March 1988 issue. For complete rules, definitions, and submission requirements, write

Gordon Bell Awards
IEEE Software
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720

IEEE **SOFTWARE**

used, but it also has the drawback that in some chapters there is not much correlation between the example and the chapter contents.

At the end of each chapter are exercises and an annotated bibliography. The exercises range from simple recall of facts stated in the chapter to open-ended design projects.

With the mention of exercises, it is worth discussing the use of this book as an adjunct to university courses. Clearly it is not a textbook in the usual sense, but it would certainly be an excellent supplementary book for an introductory course in computer science or for a variety of other courses.

A high-school student with an interest well developed in computers and electronics would certainly benefit from this book, but Slater has really written it for experienced engineers. Many of its subtleties will be lost on students who lack design experience in industrial situations.

One of the book's great strengths in a teaching context is the high quality of the editing and production. From that standpoint this is one of the best technical books I've seen in a long time, although there are curious lacunae. There are a variety of subject/verb agreement problems, and there is particular difficulty handling constructions like the one starting this sentence. I have resigned myself to seeing "different than" in well-edited books, and I have almost stopped wincing when I see "due to" introducing an adverbial clause. But I have to draw the line at "indicate if" as a synonym for "indicate whether or not," since the former can easily cause a technical reader to begin to parse the sentence incorrectly and lead to an inevitable doubletake. Furthermore, an editor with a better technical background would probably have caught lapses like "1-2 ns/ft speed" and "fast delay times." All this carping aside, however, this is a well-edited book, and no professor need feel ashamed of recommending it to students.

68000 Microcomputer Systems, Designing and Troubleshooting, Alan D. Wilcox (Prentice-Hall, Englewood Cliffs, NJ, 1987, 579 pp., $46.67)

This book on microprocessor interfacing is much more sharply focused than Slater's. Wilcox has consciously written a textbook for a specific sort of laboratory course and has provided information about how to teach from it.

I said earlier in this column that I don't read many 600-page books, and I certainly didn't read two of them this month, so what I'm going to say about this book is based on less than a complete reading of it. That "truth-in-reviewing" statement having been made, I suggest that if you have a microprocessor lab course to teach this fall, you ought to get an evaluation copy of this book and see if it meets your needs.

What I like about this book is the way it marries design theory and design practice without wasting a lot of time and energy trying to bend the latter to fit the former. The theory is presented, its relevance is made clear, it is invoked when useful, and otherwise it is kept out of the way.

Wilcox is especially interested in project planning, and this gives the book a different emphasis from other how-to books. Scheduling realistic completion dates is a skill that students would be well advised to develop early in their careers. No book will teach them this, but Wilcox's emphasis on it can't hurt.

Like Slater's book, this one devotes far less attention to software issues than to hardware issues, leaving a gap that must be filled some other way.

Next time

It should be clear to regular readers of this column that there is no relationship between what I say at this point in the column and what actually appears in the next issue. Tune in next time and find out what books and software avoid getting lost in the stack.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 177 Medium 178 Low 179

MicroNews

MicroNews features information of interest to professionals in the microcomputer/microprocessor industry. Send information for inclusion in MicroNews one month before cover date to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

Commercial era dawns for MCC

After five years of existence, the Microelectronics & Computer Technology Corporation has achieved the commercial stage.

MCC's technology helped NCR produce Design Advisor, an expert system used to design ASICs. The product, available this September as a dial-up service for NCR's ASIC customers, will be marketed as a software package in early 1988.

Although computer-aided design tools for chips are already on the market, and expert systems have been used commercially in a number of applications, Design Advisor is the first known chip-design tool to use artificial intelligence with truth maintenance technology. US Secretary of Commerce Malcolm Baldrige hailed the product as symbolic of the importance of cooperative research in a competitive international industry.

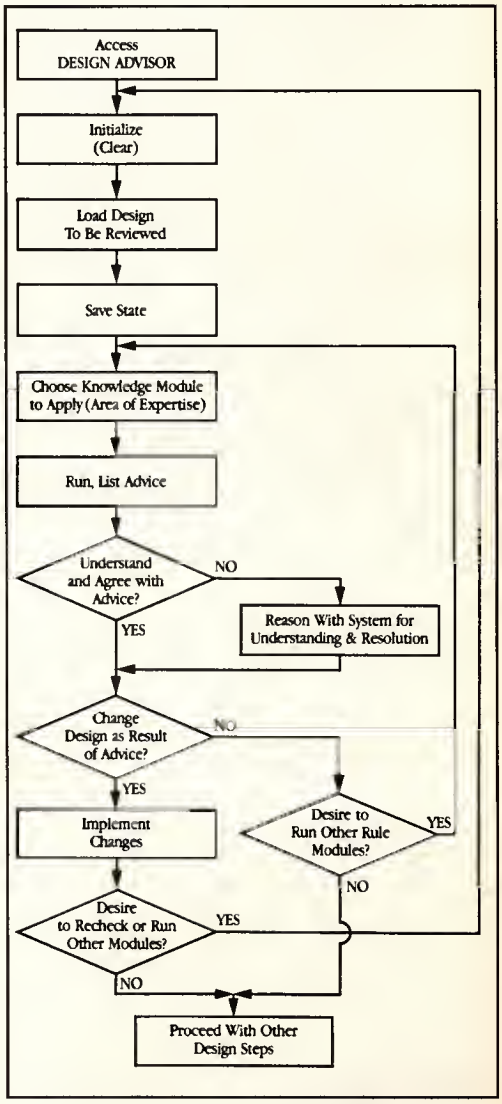
Design Advisor is based on MCC's Proteus, an expert system development facility transferred to MCC shareholders in 1985, and was produced when NCR added its knowledge base of microelectronics design. Design Advisor explains the reasoning behind its design advice and incorporates new rules and facts into the existing 1000-rule knowledge base. (See accompanying box.)

In a growing ASIC market, Design Advisor may reduce the time to market from 24 to six months and cut design costs by about 20 percent. Dial-up service fees start at $4000 per analysis, based on design complexity and number of production runs. Now running on a Symbolics processor, Design Advisor eventually will be marketed for a variety of engineering workstations.

Grant Dove, MCC's new chairman, projects that NCR's product is only the first in a series of commercial offerings expected to be announced by consortium members over the next year. Boeing reportedly plans to enhance its next 787 aircraft with circuit packaging and interconnect technology developed through MCC.

Dove, a former vice president at Texas Instruments and an engineer by training, also faces potential reorganization problems. Several companies are reported to be leaving MCC, and other shareholders, such as General Electric, are reevaluating their commitments. Dove is also interested in generating membership and incorporating new ideas.

Internally, MCC is restructuring its Advanced Computer Architecture program, which directs 15 research projects. Several projects have been dropped, and a program to study the application of superconductive materials to electronics is to begin late in 1987.



NCR Design Advisor sequence of analysis.

T1 networks gain popularity

According to a recent 195-page study from International Resource Development, large corporate telecommunications users are installing T1-based private networks by the hundreds. Researchers saw a strong trend developing toward partial or complete integration of previously separate voice and data networks.

Those with the most to gain in this movement are manufacturers of T1 multiplexers and other specialized digital equipment such as Timeplex and Network Equipment Technologies. Market saturation is expected to be several years down the road. (For an IRD forecast of private networks, see the accompanying box.)

The study projects that the winner of the FTS2000 (Federal Telecommunications System) bid will use the huge private network as a foundation for new public-switched, long-distance services.

Those interested in the $1850 study, "Private Telecommunications Networks," should contact IRD at 6 Prowitt Street, Norwalk, CT 06855; (203) 866-7800.

Forecast for private networks ($ millions)

Type of topology	1987	1989	1991	1993	1995	1997
Voice networks						
Dedicated	$5,097	$5,026	$4,917	$4,641	$4,452	$4,389
Actual	378	331	344	360	378	391
Virtual	55	116	246	301	487	621
Total voice networks	$5,530	$5,473	$5,507	$5,302	$5,317	$5,401
Data networks						
Dedicated	$1,407	$1,340	$1,225	$1,115	$1,032	$ 958
Actual	298	358	553	718	929	970
Packet	465	647	795	927	1,069	1,171
Total data networks	$2,170	$2,345	$2,573	$2,760	$3,030	$3,099
Integrated networks						
T1 services	$ 393	$ 835	$1,675	$2,191	$2,837	$3,649
T1 equipment	200	443	639	655	672	798
Digital equipment	12	29	44	50	53	55
Integrated net total	$ 605	$1,307	$2,358	$2,896	$3,562	$4,502
Total private networks	**$8,305**	**$9,125**	**$10,438**	**$10,958**	**$11,909**	**$13,002**

(Source: International Resource Development, Inc.)

Service solves schedule conflicts

Can't make it to the meeting? Now you can see it on tape.

The Los Angeles Chapter of the Computer Society of the IEEE now provides videotapes of technical meetings, seminars, and tutorials presented in the LA area. (Sorry, this service is limited to IEEE members only.)

Most tapes come with hard copies of viewgraphs used in the original presentations; many include the question-and-answer period. The initial tape fee is $20 (with figure copies if available); later tape exchange costs $5. For current library and related data, contact the Computer Society of the IEEE, LA Chapter, PO Box 1285, Pacific Palisades, CA 90272; (213) 454-1667.

Microfiche has the answer

Information Handling Service has established two microfiche resources for seekers of data on ICs and optoelectronic devices, or semiconductor devices.

The Integrated Circuit Parameter Retrieval Microfiche Service offers 146,000 technical data pages representing 380 manufacturers on 24x microfiche. Over 80,000 ICs and optoelectronic devices are listed. Users may search by type number, original circuit number, or by function/application, MIL-SPEC slash sheet number, MIL-QPL, and MIL-STD. Packaging, temperature range, and technology information are also included.

The Semiconductor Parameter Retrieval Microfiche Service, on the other hand, contains 40,000 technical data pages representing 200 manufacturers with 165,000 devices. Users may search by type number, military specification search sheet number, and by parameter. Packaging, temperature range, key and secondary parameters, and polarity are indicated.

All indexes include domestic and international sales office listings. Estimated annual subscription rates for ICPR start at $1800, with updates shipped every 30 days. SPR's minimal estimated cost is $1300 yearly, with updates shipped every 120 days.

For further details, contact Information Handling Services, 15 Inverness Way East, Englewood, CO 80150; (800) 241-7824, in Colorado (303) 790-0600, ext. 59.

Continued on p. 93

New Products

Marlin H. Mickle
University of Pittsburgh

Send announcements of new microcomputer and microprocessor products, and products for review, to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

PC neurocomputer tackles real-world problems

Hecht-Nielsen Neurocomputer Corporation has combined IBM PC ATs and compatibles with the Anza Neurocomputing Coprocessor System to form a computer capable of implementing a neural network. Anza contains software designed specifically for real-world applications.

A neural network is a computing architecture based on research into how the brain encodes and processes information. Simple networks exhibit complex behaviors such as learning, pattern recognition, and associative memory. Applications include continuous, real-time speech recognition, image recognition, processing of imperfect or inexact knowledge, and automatic extraction of knowledge from large databases.

The Anza neurocomputer is a coprocessor board usually bundled with a Zenith Model 248 PC. Anza uses a Motorola MC68020 CPU and an MC68881 floating-point coprocessor, closely coupled to 4M bytes of one-wait-state dynamic RAM. According to the company, Anza implements neural networks containing up to 30,000 processing elements (neurons) with up to 480,000 interconnects updated at 25,000 interconnects per second.

Anza software includes the User Interface Subroutine Library, which provides access to all Anza functions from within programs written in C, Pascal, Fortran, and Basic. In addition, five Basic Neware Packages are supported. These programs are generic, user-configurable implementations of a basic network paradigm. This paradigm specifies the interconnection structure of a network and the form of the differential equations that determine the behavior of individual processing elements. Each package can be customized to fit a specific application.

Hecht-Nielsen Neurocomputer Corporation's Anza system is a general-purpose neurocomputer usually bundled with a Zenith Model 248 PC and a 20M- or 40M-byte hard disk. A 1.2M-byte floppy disk and an EGA color board and monitor are included. Short and extensive training courses in neurocomputing are offered to users.

The Hecht-Nielsen Neurocomputer Corporation's Anza begins in pricing at $9500, for the board only. System pricing starts at $14,950. Two-day training courses to introduce users to neurocomputing can be attended for $1950.

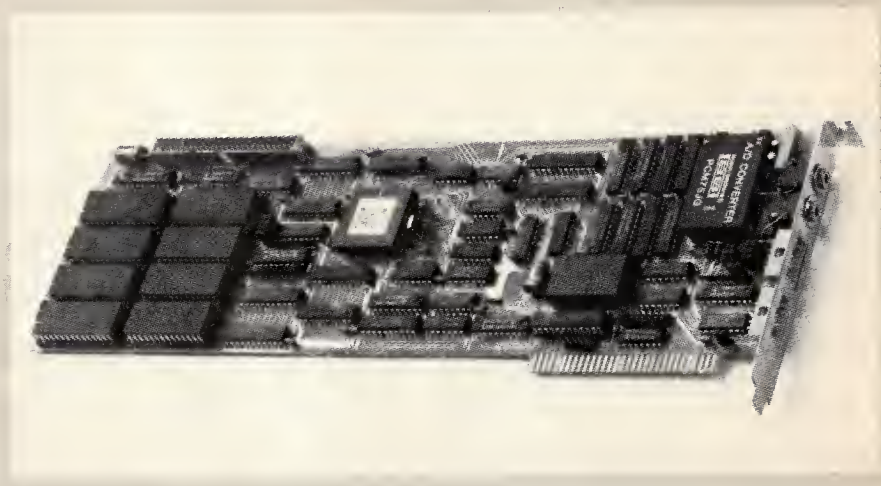
Reader Service Number 30

General-purpose DSP uses TMS320C25 processor

The TMS320C25 Development and Application System from Pacific Microcircuits is an IBM PC plug-in board usable in DSP applications in a real-time environment. The stand-alone systems with 16-bit A/D and D/A can be used to develop TMS320C25 code and eliminate the need for emulators and simulators.

The DSP system is supplied with 16 Kwords of zero-wait-state RAM, which can be expanded to 64 Kwords of program RAM and 64 Kwords of data RAM. A 12-port expansion bus provides data, address, and control lines to external hardware and supports a multiple-board environment.

Monitor software permits single-step, breakpoint, or full-speed operation. This software, resident on the PC host, can take output from the TI Macro Assembler/Linker and load it onto the board. Registers and data memory locations can be examined and modified using a variety of formats. Program memory can be examined, modified, or disassembled. Processing on the board proceeds independently of PC operations.



The Pacific Microcircuits TMS320C25 Development and Application System can be run as a stand-alone system or interfaced to external data acquisition hardware via the expansion bus. It also doubles as an application board for PC-based systems such as industrial control, patient monitoring, seismic or sonar processing, and office automation.

The TMS320C25 Development and Application System with monitor software and documentation is priced at $2595 by Pacific Microcircuits.

Reader Service Number 31

PC optical subsystem forms 200M-byte files

Optotech has enhanced its Laser Databank "plug-and-play," write-once, optical storage subsystem for PC-compatible computers. The subsystem's read-write device driver permits users to create files of up to 200 megabytes in size and use a full surface of the 5.25-inch optical disk for a single file.

The driver enables off-the-shelf PC-DOS software to write to and read from the optical drive as if it were a standard magnetic drive. Users add one line of code to the computer's system configuration file to install the driver. No supplemental hard disk is needed.

The Laser Databank contains a Model 5984 write-once optical disk drive, PC-bus controller, and PC-DOS Optical-Drive Toolkit. Optotech data files are transportable among PC, VAX, and Unix environments; PC and SCSI controllers are available.

The enhanced Optotech Laser Databank sells for $2950; volume discounts are available.

Reader Service Number 32

SCSI interface supports 16/32-bit emulators

Applied Microsystems Corp. has extended its high-speed SCSI interface option to its ES 1800 in-circuit emulators. The microprocessors supported with the option include Motorola's 68000, 68008, and 68010; Intel's 80286, 80188, and 8088; and Zilog's Z-8000.

The SCSI link handles communication between the emulator and the host. According to the company, software commands allow files to transfer in seconds. The board supports the emulator-to-host interface electrically and logically in both single-ended or differential modes. It also supports bus arbitration.

Contact Applied Microsystems for SCSI board prices.

Reader Service Number 33

CPU meets 16-bit STD-bus specifications

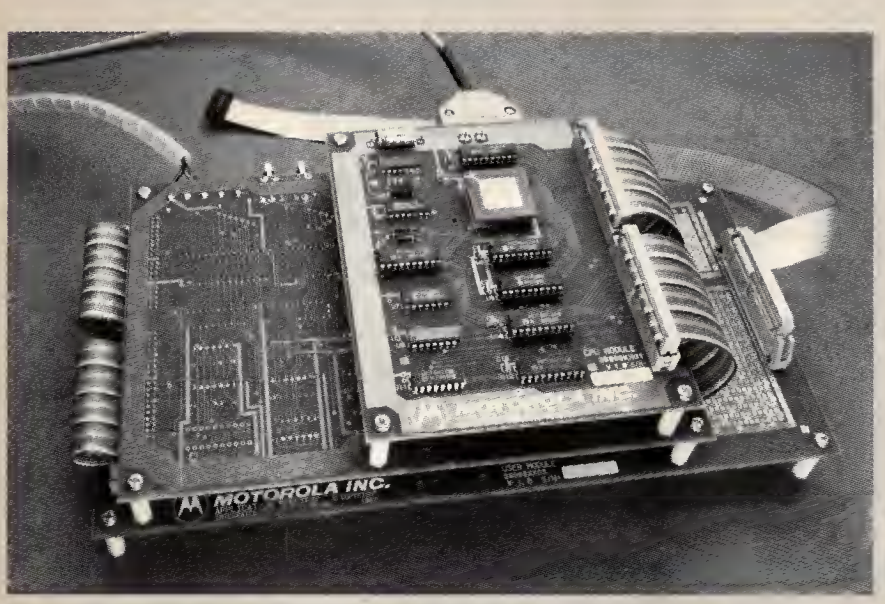
The Model 8600 from the Cubit Division of Proteus Industries is a CPU board with a 10-MHz 80186 microprocessor and a 16-bit STD bus. It is capable of expansion with additional 8- or 16-bit STD-bus boards. Monitor software links the board to an IBM PC for program development and debugging.

Two battery-backed, 32K × 8 CMOS, static RAM chips provide 64K of on-board memory, and two ROM sockets accept up to 27,512 EPROMs for a total of 128K bytes. Model 8600 also contains two 8256 serial/parallel I/O chips, which provide two RS-232C serial ports and four 8-bit parallel ports.

Cubit is selling each Model 8600 for $590, with discounts available for quantity purchases.

Reader Service Number 34

Board emulates ASICs in real time



Motorola's MPU6805 evaluation board is designed to be a basic software and hardware development tool for emulating the MPU6805 standard cell as it would be used in an ASIC device.

As a special function within its 2-Micron Double Layer Metal Standard Cell Library, Motorola is providing the EVB Module MPU6805 core. The core is an extraction from the 8-bit MC68HC05C4 microcomputer unit and executes an identical instruction set.

With the EVB Module designers can perform real-time emulation of application-specific ICs, evaluating target system performance and debugging user-assembled code through device simulation and emulation. Designers can also use the evaluation board to implement software breakpoints, perform single-step tracing, and examine the memory and registers of the user's program.

All normal user-provided resources such as RAM, I/O peripherals, and EPROM are omitted from the board; the designer builds or provides these features on a separate wire-wrap user board. This board holds the peripheral circuits, which together with the core cell make up the balance of logic and memory contained on the semicustom chip being emulated.

The EVB Module requires user-supplied power of +5, +12, and −12 Vdc and an RS-232C-compatible terminal or host computer for complete operation.

Priced at $500, the MPU6805 evaluation board is available immediately from Motorola.

Reader Service Number 35

Generic BIOS chip set uses 286 code

Electro Design's Mighty BIOS combines firmware with 286 BIOS, debug, and set-up programs to expand the use of IBM PC ATs and compatibles for programmers. The programs substitute for the Basic language in these systems. According to the company, the chip set has run at 16-MHz clock speeds.

The installation set-up program contains features found on the diagnostic disks of a variety of machines; it also displays the events in progress as these functions are checked for status and tested for adequacy. Changes or additions to the program can be made by using four arrow keys on the keyboard.

The 286 debug program can be loaded into RAM as an optional feature to analyze, move by bit or block, change, save, or delete data prior to reentry into the operational program.

A single Electro Design Mighty BIOS chip set costs $139; OEM discounts are available.

Reader Service Number 36

Chip set replaces PC/AT devices

The Application Specific Logic Products Division of VLSI Technology has introduced a set of five chips that reduces the device count on the motherboards of IBM PC/ATs from 110 industry-standard devices to 16 ICs, excluding memory devices. The chip set supports 1M-bit DRAMS and can be used in systems with clock speeds up to 12 MHz.

Included in the chip set are the VL82-C100 peripheral controller; VL82C101 system controller; VL82C102 memory controller, which generates the row address strobe and column address strobe necessary to support the PC/AT DRAM; VL82C103 address buffer; and VL82C104 data buffer. The advanced CMOS devices are available in a JEDEC-standard, 84-pin PLCC package.

Sample kits of the five-chip set are available for $225, until the third quarter when production pricing of $68.85 in 10,000-set quantities goes into effect.

Reader Service Number 37

Intel adds to UPI family

Intel Corp. has introduced the plastic 8742AH to its UPI 42 family of universal peripheral interfaces. These keyboard interfaces are used in IBM PS/2 Models 50/60/80, PC ATs, and printer and display controllers. The P8742AH uses an EPROM die housed in a windowless, plastic package capable of use by automatic insertion equipment.

According to the company, the Intelligent Programming algorithm used in the P8742AH makes it 20 times faster to program than the current 8742. Other features include an 8749 microcontroller core, 2K-byte EPROM, 256-byte RAM, a security bit for code-theft protection, 18 programmable I/O pins, and an 8-bit timer/counter. The P8742AH operates at 12 MHz and is code-compatible with family products.

Intel prices the P8742AH at $12 for a 40-lead plastic DIP package and at $14 for a 44-lead PLCC, in lots of 100.

Reader Service Number 38

PS/2 storage, backup systems available

Data storage and backup systems compatible with the IBM Personal System/2 and its predecessors have been announced by Mountain Computer. The line is designed to operate under DOS 3.3 and the OS/2 operating system.

Systems include the Model 30, a 40M-byte, TD4000 external tape drive; the FileSafe Series 7000 tape storage units with capacities of 60M and 120M bytes; the self-contained, 20M- or 30M-byte, 3.5-inch DriveCard hard disk with controller plug-in card; and the MicroBernoulli 20M-byte removable cartridge drive. According to the company, all are available for shipment.

Contact Mountain Computer for prices.

Model 30, Reader Service Number 39
FileSafe, Reader Service Number 40
DriveCard, Reader Service Number 41
MicroBernoulli, Reader Service Number 42

Silicon compilers improved

Silicon Compiler Systems has enhanced its Generator Development Tools system by adding a generator-calling feature, more design libraries, and a verification feature for faster simulation. GDT is a set of tools that enables IC designers to build and verify reconfigurable circuit generators (cell or silicon compilers).

Another enhancement, a mouse-controlled interface, accesses GDT features while hiding Unix complexity; permits multiple windows for editing, user and system comments, and a history transaction file; and provides a text editor.

The generator-calling feature enables designers to interactively gain access to generator program libraries and incorporate them into the IC design database on which they're working. A simplified verification feature permits designers to use a single multilevel, mixed-mode simulator for all stages of the IC design process. Additionally, modes can be mixed during one simulation. Users can pan and zoom over results and make voltage and timing measurements directly from the waveforms.

The software enchancements from Silicon Compiler Systems begin in price at $49,500; pricing depends on the configuration.

Reader Service Number 43

Expert system controls factory automation

A dedicated, microcomputer-based expert system designed to detect problems, diagnose their probable cause, and assist in solving these problems in factory automation applications has been introduced by UMECORP. Expert Controller can function independently of an operator; it can also guide operators through critical restart procedures.

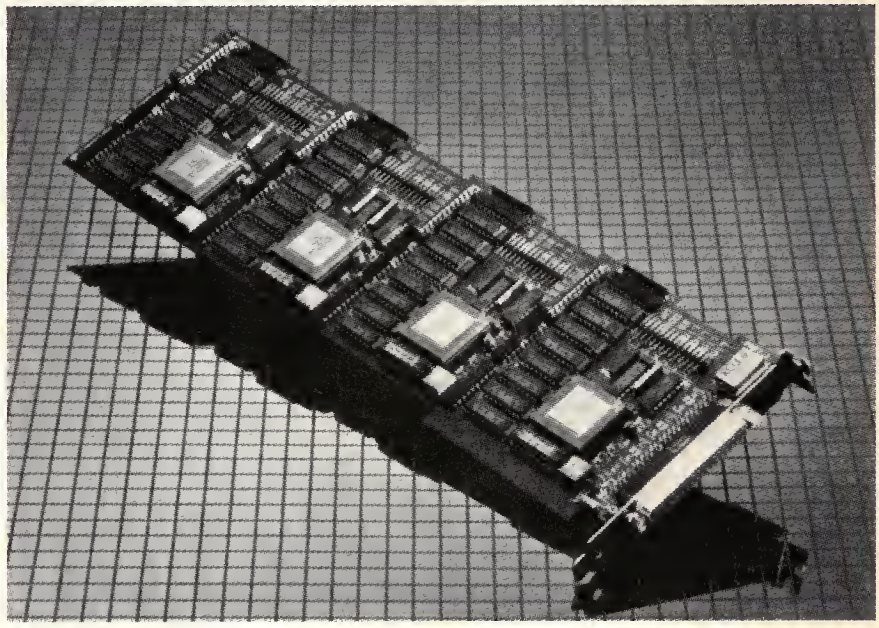
The programmable Expert Controller consists of a host-independent, asynchronous-processing microcomputer using CMOS technology and proprietary software with a natural-language development interface. About half the size of a briefcase, the computer can be configured as a stand-alone system or combined with several units in a rack mount with RS-485 multidrop for larger scale, distributed artificial intelligence applications.

Expert Controller includes a battery-backup static RAM or EPROM for knowledge bases, a built-in power supply, and environmentally resistant industrial packaging. The system allows users to develop modular knowledge bases and test them off line before connecting them to operating industrial equipment.

The UMECORP Expert Controller sells for $5500 with multiunit discounts.

Reader Service Number 44

Transputer line announced



Computer System Architects combines parallel processing hardware and software in its Parallel Transputer series for use with IBM PCs and compatibles.

Computer System Architects is offering a family of flexible hardware and software experimentation tools designed for development projects in parallel processing areas. The board series, built around the 32-bit Inmos Transputer and Transputer Development System for the Occam language, can be plugged into an IBM PC, XT, AT, or compatible.

The Parallel Transputer Series PART.1 board contains four Transputers, each with 256K bytes of DRAM storage. Users can configure all line interconnections to accommodate various Transputer interconnect topologies. PART.1 is appropriate for parallel Transputer evaluation and concurrent processing problem solution. Companion boards contain larger memory and special interfaces to other processors and to secondary store.

Contact Computer System Architects for pricing.

Reader Service Number 45

No-wait-state, 20-MHz board speeds PCs

CSS Laboratories is offering its 16/20-MHz 386 Performer motherboard to upgrade performance of IBM PCs and compatibles. According to the company, a Norton Utility benchmark run on the zero-wait-state board shows a rating of 18.7 at 16 MHz.

The AT-size version of the 80386 is keyboard selectable for 8 MHz or 16 MHz and features one 32-bit memory slot, five 16-bit, and two 8-bit expansion slots. On-board, 1M-byte, static column RAM can be upgraded to 2M bytes; the 16-MHz 386 board can be upgraded to 20-MHz by adding both memory and microprocessor. An adapter board socket for the Weiteck extended coprocessor permits CAE/CAD applications to be processed.

The list price for the 386 Performer from CSS Laboratories is $1980.

Reader Service Number 46

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 180 Medium 181 Low 182

MicroNews

Continued from p. 88

Standards breakthroughs announced

A new committee to produce urgently needed international standards for open systems interconnection, microprocessor systems, computer graphics, and data communications will meet in Tokyo this November 17-20.

Formation of the ISO/IEC Joint Technical Committee I has been approved by the councils of the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), while the American National Standards Institute (ANSI) will administer the international secretariat of JTC 1, Information Technology.

ANSI, a participating member of the committee, is organizing a US technical advisory group (TAG) to obtain US viewpoint and provide representation at international meetings.

JTC 1 will integrate the activities of ISO Technical Committee 97, Information Processing Systems; IEC Technical Committee 83, Information Technology Equipment; and IEC Subcommittee 47B, Microprocessor Systems.

Progress toward a multilingual code standard for worldwide communication was announced after a recent West Berlin meeting of an ISO international subcommittee on character sets and information coding. After several years of work, a specific structure was selected for the development of a multiple octet code to serve as basis for the standard. The code will be composed of characters from existing one- and two-byte national and international character and control code sets with space for future additions.

Also of note is the recent publication of the first American National Standard to result from the work of ANSI-Accredited Committee on Telecommunications, T1. ANSI T1.101-1987 describes and specifies types of digital network interfaces carrying synchronization reference, signal specification for DS1 and 2.048-MHz reference signals, and performance specifications for synchronization parameters at network and equipment interfaces.

Copies are available for $10 each from ANSI's sales department, 1430 Broadway, New York, NY 10018.

RISC and GaAs IC trends diverge

A new report asserts that RISC-based computer system sales could reach $16.25 billion by 1991.

According to "RISC: Applications, Strategies and Markets," released by Market Intelligence Research Company, RISC-based superminis represented 41 percent of the total $540-million RISC market in 1985. Between 1985 and 1991, advanced PCs are anticipated to provide the largest market growth.

Not so impressive is the GaAs IC market, which is existing on a commercial level solely because of military interest, according to "The GaAs IC and Wafer Markets" by MIRC. Reasons given for lackluster GaAs market performance range from low levels of integration of components on a chip due to SSI and MSI technology to longer lag times to chip insertion in the military sector.

These reports are for sale for $995 each from Market Intelligence Research Company, 4000 Middlefield Road, Palo Alto, CA 94303.

New IEEE Tools Task Force invites participation

The Computer Society's Technical Activities Board has announced the formation of a task force chartered to provide tool-related services to individuals and organizations developing hardware and software products. New participants are being sought for working groups within the task force to pursue two startup development activities: hardware and software tool interface standards, and a tools inventory service.

Standards activities now underway include

1. the interface between requirements definition tools and design tools,
2. the interface between requirements definition and testing tools, and
3. an engineering database scheme with which other tools may interface.

In the proposed inventory service, vendors would enter data about available tools directly into a main database via dial-up telephone service. Inventory service users could dial up the system and query it for specific tool information by entering requirements into a fill-in-the-blank screen. Names and descriptions of the vendor-reported tools that matched the query would be provided to the user on screen or in a hard-copy report.

For more information, contact Robert Poston, Programming Environments, 4043 State Hwy. 33, Tinton Falls, NJ 07753; Compmail + r.poston.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

High 183 Medium 184 Low 185

Product Summary

Marlin H. Mickle
University of Pittsburgh

For more information, circle the appropriate Reader Service Number on the Reader Service Card at the back of this magazine.

Manufacturer	Model	Comments	Rs No.
Boards			
The Complete PC	CAM voice message board	MS-DOS PC voice-messaging coprocessor operates concurrently with most PC applications and provides personal voice mail, message handling, and basic functions for telephone answering machines. The memory-resident, add-in board is available from the keyboard, locally attached telephone, or remote touch-tone phones. $349.	60
Qualogy, Inc.	QPC-5121	Single-board, 80286-based microcomputer replaces the IBM PC AT and carries 1M bytes of RAM and 128K bytes of EPROM on board and encoded with the IBM-compatible Phoenix BIOS. Speaker, keyboard, and reset ports are included. Also features a controller for two floppy-disk drives, SCSI controller, and CMOS parts. $1695 each with diagnostics diskette, 30-45 days ARO.	61
Vector Electronic Company	Model 4613-4	Prototyping boards for IBM PCs and compatibles allow designers and engineers to build circuits using 8/16-bit microcomputer chips. An etched circuit pattern on the boards contains drilled, 0.055-inch-diameter holes spaced on a 0.1-inch × 0.1-inch grid. The voltage/ground plane series permits a T123 eyelet to be placed in any circuit board hole so leads can be connected to the circuit plane with solder. $42.30 for quantities 1-4.	62
Software			
hDC Computer Corporation	EGA-16+ driver	Microsoft Windows display driver uses the 82C435 Enhanced Graphics Controller to provide 640 × 480 resolution and up to 16 colors. Advanced capabilities support Micrografx Draw and Aldus PageMaker applications. Package includes the driver and a royalty-free license for distribution with a vendor's EGA product.	63
Whitesmiths	MICSIM	Microprocessor simulator mimics a dedicated microprocessor on a VAX, IBM PC, Sun, or other host. Users can test, debug, and monitor the performance of embedded programs they write in Pascal, C, or Assembler. All versions use the same command language.	64
Systems			
Digital Vision	Computereyes/ TelePaint 2.3	Video digitizing system for IBM PCs and compatibles captures and edits high-resolution images before integration into desktop publishing, image databases, presentation packages, and printing applications. The mouse-driven TelePaint graphics package supports on-screen drawing, editing, cropping, text addition, and copy-and-paste capabilities. Computereyes permits brightness and contrast adjustments, dither shading, coloring, and titling. A driver for the Polaroid Palette permits 35-mm slides to be made from digitized images. $299.95.	65

THE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Executive Committee

President: Roy L. Russo*
IBM T.J. Watson Research Center
PO Box 218, Route 134
Yorktown Heights, NY 10598
(914) 945-3085

President-Elect: Edward A. Parrish, Jr.*

Vice Presidents
Education: Michael C. Mulder (1st VP)*
Technical Activities: Kenneth R. Anderson (2nd VP)*
Area Activities: Willis K. King†
Conferences and Tutorials: James H. Aylor†
Membership and Information: Merlin G. Smith†
Publications: J.T. Cain†
Standards: Helen M. Wood

Secretary: Duncan H. Lawrie
Treasurer: Joseph E. Urban
Past President: Martha Sloan*
Division V Director: Martha Sloan
Division VIII Director: H. Troy Nagle†
Executive Director: T. Michael Elliott†

*voting member of the Board of Governors
†nonvoting member of the Board of Governors

Board of Governors

Term Ending 1987	Term Ending 1988
Barry W. Boehm	Mario R. Barbacci
Paul L. Borrill	Victor R. Basili
Glen G. Langdon, Jr.	Lorraine M. Duvall
Duncan H. Lawrie	Michael Evangelist
Susan L. Rosenbaum	Allen L. Hankinson
Bruce D. Shriver	Laurel Kaleda
Harold S. Stone	Ted Lewis
Wing N. Toy	Ming T. Liu
Helen M. Wood	Earl E. Swartzlander, Jr.
Akihiko Yamada	Joseph E. Urban
Oscar N. Garcia†	

Next Board Meeting

8:30 a.m.-5 p.m., October 30, 1987,
Loew's Anatole Hotel, Dallas

Senior Staff

Executive Director: T. Michael Elliott
Editor and Publisher: True Seaborn
Director, Computer Society Press: Chip G. Stockton
Director, Conferences: William R. Habingreither
Director, Finance and Administration: Mary Ellen Curto

Offices of the Computer Society

Headquarters Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1903
General Information: (202) 371-0101
Publications Orders: (800) 272-6657
Telex: 7108250437 IEEE COMPSO

Publications Office
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
Membership and General Information: (714) 821-8380

European Office
13, Avenue de l'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 770-21-98
Telex: 25387 AVVALB

Purpose

The Computer Society strives to advance the theory and practice of computer science and engineering. It promotes the exchange of technical information among its 90,000 members around the world, and provides a wide range of services which are available to both members and nonmembers.

Membership

Members receive the highly acclaimed monthly magazine *Computer*, discounts on all society publications, discounts to attend conferences, and opportunities to serve in various capacities. Membership is open to members, associate members, and student members of the IEEE, and to non-IEEE members who qualify as affiliate members of the Computer Society.

Publications

Periodicals. The society publishes six magazines (*Computer, IEEE Computer Graphics and Applications, IEEE Design & Test of Computers, IEEE Expert, IEEE Micro, IEEE Software*) and three research publications (*IEEE Transactions on Computers, IEEE Transactions on Pattern Analysis and Machine Intelligence, IEEE Transactions on Software Engineering*).

Conference Proceedings, Tutorial Texts, Standards Documents. The society publishes more than 100 new titles every year.

Computer. Received by all society members, *Computer* is an authoritative, easy-to-read monthly magazine containing tutorial, survey, and in-depth technical articles across the breadth of the computer field. Departments contain general and Computer Society news, conference coverage and calendar, interviews, new product and book reviews, etc.

All publications are available to members, nonmembers, libraries, and organizations.

Activities

Chapters. Over 100 regular and over 100 student chapters around the world provide the opportunity to interact with local colleagues, hear experts discuss technical issues, and serve the local professional community.

Technical Committees. Over 30 TCs provide the opportunity to interact with peers in technical specialty areas, receive newsletters, conduct conferences, tutorials, etc.

Standards Working Groups. Draft standards are written by over 60 SWGs in all areas of computer technology; after approval via vote, they become IEEE standards used throughout the industrial world.

Conferences/Educational Activities. The society holds about 100 conferences each year around the world and sponsors many educational activities, including computing sciences accreditation.

European Office

This office processes Computer Society membership applications and handles publication orders. Payments are accepted by cheques in Belgian francs, British pounds sterling, German marks, Swiss francs, or US dollars, or by American Express, Eurocard, MasterCard, or Visa credit cards.

Ombudsman

Members experiencing problems — late magazines, membership status problems, no answer to complaints — may write to the ombudsman at the Publications Office.

Information

Use the Reader Service Card to obtain the following material:

- Membership information and application (RS #202)
- Publications catalog (proceedings, tutorials, standards) (RS #201)
- Periodicals subscription application/information for individuals (members, sister-society members, others) (RS #200)
- Periodicals subscription application/information for organizations (libraries, companies, etc.) (RS #199)
- List of awards and award nomination forms (RS #198)
- Technical committee list and membership application (RS #197)
- Directory of officers, board members, committee chairs, representatives, staff, chapters, standards working groups, etc. (RS #196)

Advertiser Index

FOR DISPLAY ADVERTISING INFORMATION CONTACT

Southern California and Mountain States: Richard C. Faust Company, 24050 Madison Street, Suite 100, Torrance, CA 90505; (213) 373-9604.
Northern California and Pacific Northwest: Don Farris Company, 161 W. 25th Ave., #102B, San Mateo, CA 94403; (415) 349-2222.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070; (408) 741-0354.
East Coast: Hart Associates, Inc., P.O. Box 339, 42 Lake Blvd., Matawan, NJ 07747; (201) 583-8500.
New England: Arpin Associates, P.O. Box 227, 51 Colchester, Weston, MA 02193; (617) 899-5613.
George Watts, III, 4 Conifer Dr., Wilbraham, MA 01095; (413) 596-4747.
Midwest: Thomas Knorr, Knorr MicroMedia, Inc. 333 North Michigan Ave. Chicago, IL 60601; (312) 726-2633.
Southwest: The House Company, 3000 Weslayan, Suite 345, Houston, TX 77027; (713) 622-2868.
Southeast/Telemarketing: Kay Young and Assoc., 2081 Business Center Dr., Suite 180, Irvine, CA 92714; (714) 551-4924.
Europe: Heinz J. Gorgens, Parkstrasse 8a, D-4054 Nettetal 1-Hinsbeck (F.R.G.); (0 21 53) 8 99 88.

Advertising Director: Dawn Peck, IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-3240, 821-8380.

For production information, conference or classified advertising, contact Heidi Rex or Marian Tibayan. IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-8380.

Product Index

COMING in the October issue of *IEEE Micro!*

Look for the following topics in our special European issue:

☐ **Western Europe's industry strategy**
Already a technological giant, Western Europe is successfully improving its technology base and raising its global competitive level through innovative cooperative programs.

☐ **Advanced architectures**
Four articles sample recent Western European research into advanced architectures. Developed in industrial research labs, these works are strongly influenced by the requirements of practical application.

☐ **The Inmos IMS T800 Transputer and its use in close connection with programming in Occam** (from the United Kingdom)

☐ **A decentralized object-oriented machine and system architecture** (from The Netherlands)

☐ **A distributed system architecture for telecommunications** (from Italy)

☐ **A survey of solutions to fault-tolerance problems in real-time data processing systems** (from Switzerland)

ADDED HIGHLIGHT: This special issue carries the official- and only- program for the BUSCON UK Bus/Board Users Show and Conference in Heathrow, London, October 13-14.

READER SERVICE CARD

IEEE MICRO

August 1987 issue
(card void after February 1988)

Name_____

Title_____

Company_____

Address_____

City_____ State___ ZIP_____

Country_____ Phone (____) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **1**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	190
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

READER SERVICE CARD

IEEE MICRO

August 1987 issue
(card void after February 1988)

Name_____

Title_____

Company_____

Address_____

City_____ State___ ZIP_____

Country_____ Phone (____) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **2**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	190
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

SUBSCRIPTION CARD

SUBSCRIBE TO IEEE MICRO

☐ YES, sign me up! ☐ Renew my subscription!

If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$17** for a year's subscription (6 issues). Subscriptions are annualized with membership. For orders submitted March through August, please pay half the given full-year fee for a half-year's subscription.

Society:_____ IEEE membership no:_____

☐ YES, sign me up! ☐ Renew my subscription!

If you are a member of ACM, NSPE, ACS, IEE (UK), SCS, IPSJ, IECEJ, or any other technical society, pay the sister society rate of only **$25** for a year's subscription (6 issues).

Society:_____ Mem. no. (if any):_____

_____
Full Signature *Date*

Name

Street

City

State/Country *ZIP/Postal Code*

☐ Payment enclosed

☐ Charge to ☐ Visa ☐ MasterCard ☐ AmEx

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐
Charge Card Number

Mo. Yr.
☐☐ ☐☐
Exp. Date

M887

Charge orders also taken by phone:
(714) 821-8380 8:00 a.m. to 5:00 p.m. Pacific time
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720

Editorial comments

I liked: _____

_____

_____

_____

_____

I disliked: _____

_____

_____

I would like to see: _____

_____

_____

_____

_____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

Editorial comments

I liked: _____

_____

_____

_____

_____

I disliked: _____

_____

_____

I would like to see: _____

_____

_____

_____

_____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Computer Society of the IEEE
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-9970

1988 - 1989

INTERNATIONAL EDITORIAL CALENDAR

1988

FEBRUARY
General Interest
An inside look at the most important developments in microprocessors and microcomputers

APRIL
Special Far East Issues
The latest technology in Japan, Korea, Taiwan, and the Pacific Basin—microprocessors, support devices, new systems, standards, and innovative applications
Submit by: 9-15-87 and 9-15-88

JUNE
VMEbus Software, ASICs, and Distributed Systems
Submit by: 11-15-87

AUGUST
Fault Tolerance
Analyses of devices and systems for handling and correcting errors
Submit by: 1-15-88

DECEMBER
Digital Signal Processing
Updates this fast-moving technology: newer, faster, smaller?
Submit by: 5-15-88

1989

FEBRUARY
Applications
The Racer's Edge: Advanced applications using microprocessors
Submit by: 7-15-88

JUNE
High-Performance Processors
Current CISCs and RISCs compared
Submit by: 11-15-88

AUGUST
Memory Technology, CAD, and CIM
Submit by: 1-15-89

OCTOBER
Special European/Near East Issues
Annual display of the most exciting microelectronic technology on and off the continents
Submit by: 3-15-88 and 3-15-89

DECEMBER
General Interest
Come to us with a micro-related topic
Submit by: 5-15-89

IEEE MICRO

IEEE Micro helps designers and users of microprocessor and microcomputer systems explore the latest technologies to achieve business and research objectives.

Feature articles in **IEEE Micro** are original works relating to the design, performance, or application of microprocessors and microcomputers. Tutorial material, industry views, and standards developments are also published.

AO CLOSING OATE: 1st of month preceding issue [Jan. 1st for February issue]. Contact Advertising Director Dawn Peck, Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578; [714] 821-8380.*

HOW TO SUBMIT AN ARTICLE TO IEEE MICRO

Prospective contributors should request Author Guidelines from:

James J. Farrell III
Editor-in-Chief, IEEE Micro
VLSI Technology Inc.
8375 South River Parkway
Tempe, AZ 85284
[602] 752-6222

All manuscripts are subject to a peer-review process consistent with professional-level technical publications.

IEEE Micro is a bimonthly publication of the Computer Society of the Institute of Electrical and Electronics Engineers.

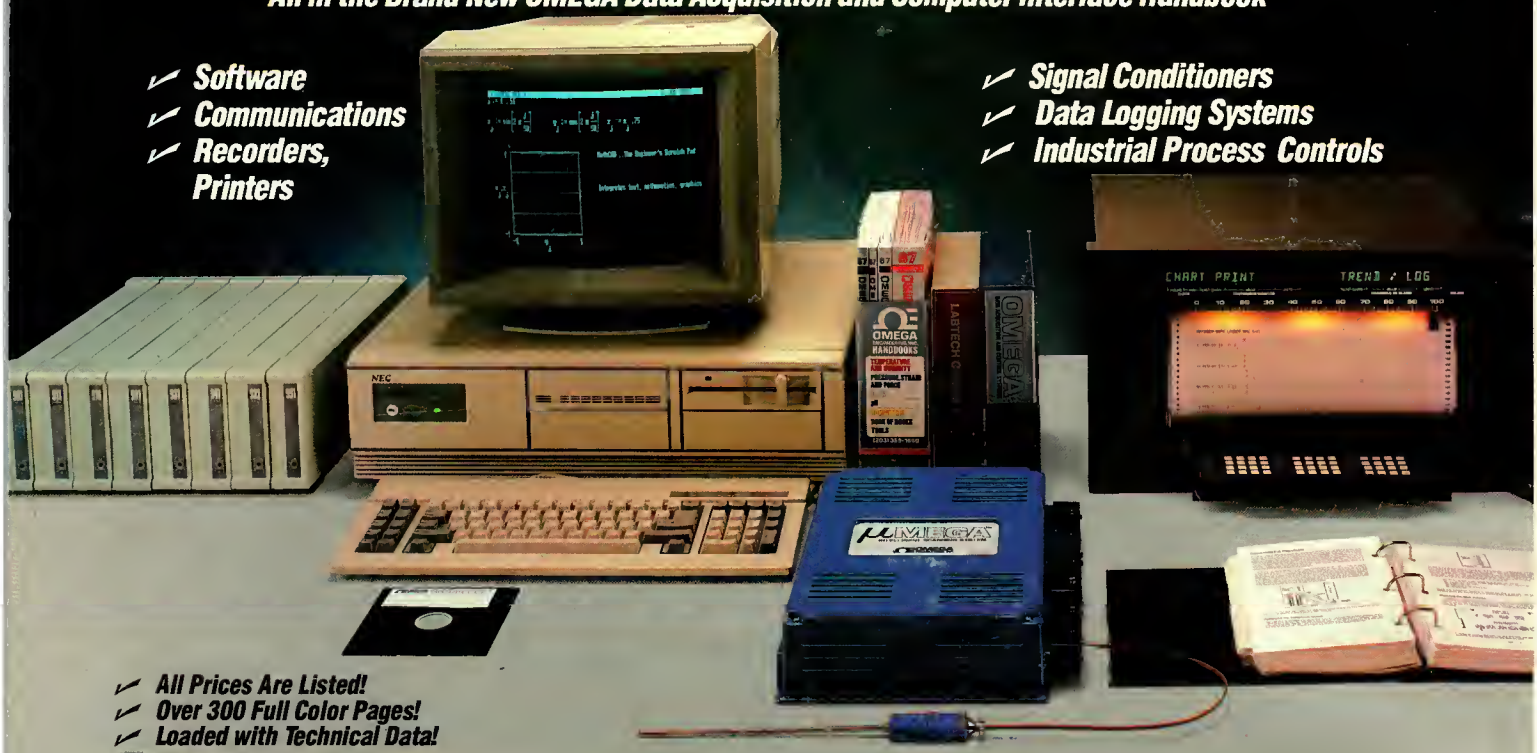
*Articles may change. Please contact editor to confirm.

OMEGA® is...

Complete Data Acquisition Systems for Measurement and Control

All in the Brand New OMEGA Data Acquisition and Computer Interface Handbook

- *Software*
- *Communications*
- *Recorders, Printers*
- *Signal Conditioners*
- *Data Logging Systems*
- *Industrial Process Controls*



- *All Prices Are Listed!*
- *Over 300 Full Color Pages!*
- *Loaded with Technical Data!*

OMEGA®

An OMEGA Group Company

One Omega Drive, Box 4047, Stamford, CT 06907
Telex 996404 Cable OMEGA FAX (203) 359-7700

In a Rush For Your FREE Handbooks?

DIAL (203) 359-RUSH
(203) 359-7874

©COPYRIGHT 1987, OMEGA ENGINEERING, INC. PRINTED IN USA

Reader Service Number 1

IEEE MICRO

DECEMBER 1987

GaAs Technology Matures

One-chip RISCs zoom to the forefront

THE COMPUTER SOCIETY OF THE IEEE

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE MICRO

Volume 7 Number 6 (ISSN 0272-1732) December 1987

Published by the Computer Society of the IEEE

On the Cover

Read more about the two gallium arsenide chips pictured on the cover by turning to pp. 3 and 8.

Cover illustration:
Jim Brummond

STAFF

Editor and Publisher
True Seaborn

Assistant Publisher
Douglas Combs

Managing Editor
Marie English

Assistant Editor
Christine Miller

Assistant to the Publisher
Pat Paulsen

Advertising Director
Dawn Peck

Art Director
Jay Simpson

Design and Production
Tricia Hayden

Membership Manager
Christina Champion

Circulation Manager
Paul Zive

Advertising Coordinators
Heidi Rex, Marian Tibayan

Reader Service
Marian Tibayan

Departments

Feature Articles

Circulation: IEEE MICRO (ISSN 0272-1732) is published bimonthly by the Computer Society of the IEEE: IEEE Headquarters, 345 East 47th St., New York, NY 10017; Computer Society West Coast Office, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Annual subscription: $17 in addition to Computer Society or any other IEEE society member dues; $25 for members of other technical organizations. This journal is also available in microfiche form.

Postmaster: Send address changes and undelivered copies to IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. Second class postage is paid at New York, NY, and at additional mailing offices.

Copyright and reprint permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US Copyright Law for private use of patrons: those post-1977 articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 29 Congress St., Salem, MA 01970. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint, or republication permission, write to Reprints, IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All rights reserved. Copyright © 1987 by the Institute of Electrical and Electronics Engineers, Inc.

Editorial: Unless otherwise stated, bylined articles and descriptions of products and services in New Products, Product Summary, MicroReview, MicroNews, and MicroLaw, reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the Computer Society of the IEEE. Send editorial correspondence to IEEE MICRO, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578. All submissions are subject to editing for style, clarity, and space considerations. IEEE MICRO subscribes to The Computer Press Association's code of professional ethics.

BPA

IEEE Micro

Editor-in-Chief:
James J. Farrell III
VLSI Technology Incorporated*

Associate Editor-in-Chief:
Joe Hootman
University of North Dakota

Editorial Board
Shmuel Ben-Yaakov
Ben Gurion University of the Negev
Dante Del Corso
Politecnico di Torino, Italy
John Crawford
Intel Corporation
Stephen A. Dyer
Kansas State University
K.-E. Grosspietsch
GMD, Germany
David B. Gustavson
Stanford Linear Accelerator Center
Victor K.L. Huang
AT&T Information Systems
Barry W. Johnson
University of Virginia
David K. Kahaner
National Bureau of Standards
Jay Kamdar
National Semiconductor Corporation
G. Jack Lipovski
University of Texas
Kenneth Majithia
IBM Corporation
Richard Mateosian
Marlin H. Mickle
University of Pittsburgh
Varish Panigrahi
Digital Equipment Corporation
Ken Sakamura
University of Tokyo
Michael Smolin
Smolin & Associates
Richard H. Stern
Yoichi Yano
NEC Corporation

Magazine Advisory Committee
Michael Evangelist (chair),
Vishwani D. Agrawal, James J. Farrell III,
Ted Lewis, David Pessel, True Seaborn,
Bruce D. Shriver, John Staudhammer

Publications Board
J.T. Cain, (chair), Vishwani D. Agrawal,
J. Richard Burke, Gerald L. Engel,
Michael Evangelist, James J. Farrell III,
Lansing Hatfield, Ronald G. Hoelzeman,
Ted Lewis, Ming T. Liu, Ed Nahouraii,
David Pessel, C.V. Ramamoorthy,
Vincent D. Shen, Bruce D. Shriver,
John Staudhammer, Steven L. Tanimoto

*Submit six copies of all articles and special-issue proposals to James J. Farrell III, 8375 South River Parkway, Tempe, AZ 85284; (602) 752-6222; Compmail + j.farrell.

From the Editor-in-Chief

A very large number of comments appeared in the mailbag this month. As usual, I am grateful for your comments and suggestions. Our industry changes so quickly that communication from readers worldwide is extremely important in keeping *IEEE Micro* relevant. The reader response cards travel here from all continents except Antarctica.

The recent Computer Society magazine readership survey indicated that *IEEE Micro* had the most loyal readers (about 89 percent intend to renew this year). Albeit we won distinction in this category over the other CS magazines by a very small margin, this is still very gratifying. While we are vigorously looking for new subscribers, we are very attentive to the comments of those currently subscribing as well. When I see a well-written technical article submitted

The mailbag

"All of the articles are well written and simple to read...." R.A., Como, Italy

"I find it amazing that the June issue arrived two weeks BEFORE the April issue." H.D.A., Wellington, New Zealand (The June issue was mailed precisely two months after the April issue. I suspect that your April issue spent a good deal of time stuck in the bottom of a mailbag somewhere.—J.F.)

"I liked the TRON project—good stuff...BTRON is the way to go...Look out, DOS-TRON is coming!" J.N., Suva, Fiji Islands

"More articles on microprocessor applications for hydraulic/pneumatic circuitry." F.J.B.B., New London, CT (Being familiar with the New London area, I suspect that F.J.B.B.'s work involves nautical electronics.—J.F.)

"I liked the standards report." Bothell, WA

"I would like to see a bit more depth and detail in MicroLaw column." T.P.G., Torrance, CA (Richard Stern has written a detailed and extensive book on semiconductor copyright law, if you wish further study.—J.F.)

"I liked Clipper, Am29000, Micro-Review, and MicroLaw. I would like more tutorials on microprocessors and VLSI." K.W., Nashville, TN

"Compare the 80387 to the Weitek 80387 replacement....(*Micro*) New Products Department is a wimpy substitute of a concept. For the real thing, check out (several commercial trade publications listed)." R.G., Randolph, NJ (Most comments indicate that our new products are right on target. We get the same news releases everyone else gets. Being bimonthly, our major problem in this department is timeliness.—J.F.)

"I liked little—collection of pseudo-intellectuals. I disliked extreme theory—wrong magazine. I would like to see clean applications articles by American authors." (origin withheld) (*IEEE Micro* reviews all material for publication without regard for the sex, race, religion, handicap, or national origin of the author.—J.F.)

"I liked New Products section, IEEE standards news." M.B., Brisbane, Australia

"I liked the new generation of microprocessors, introduction to the Clipper architecture, the 80387 and its applications." W., Vernon, France

"'System Considerations in the Design of the Am29000' was superb. I would like to see more articles such as the Am29000. How about articles on MIPS-X, Acorn's RISC processor, etc.?" V.L.D., Bangalore, India (Your suggestion is well taken. We have scheduled a special issue on this for June 1989, but perhaps we should address some RISC processor issues sooner. – J.F.)

"I liked the lack of HYPE in the microprocessor articles; reveals the honest

from Saudi Arabia receive complimentary comments from such geographically diverse locations as Mexico, the United Kingdom, and the Fiji Islands, I know that *IEEE Micro* is serving its readers.

Final note: It is with great sadness that we note the death of W.H. Brattain, one of the inventors of the transistor and a giant of science. He will be missed and remembered.

Best regards,

Jim Farrell

excitement of this field. THANK YOU. I would like to see monthly issues, please." R.J., Louisville, KY (I couldn't agree more; unfortunately, monthly issues are not presently in our plans.—J.F.)

"I would like to see more on VLSI and algorithms for signal processing...." S.G., Newcastle upon Tyne, UK

"I liked 'A Synthetic Instruction Mix.' It was a very good article that helps in choosing a microprocessor...more articles on mass use (e.g., vending) machines." M.J.H., Vina DelMar, Chile

"I liked the speech analysis article." J.N., Suva, Fiji Islands

"I liked the article on speech analysis and synthesis." R.H.R., Manchester, UK

"All of the articles in this (June) issue were excellent." J.G.P., Montreal, Canada

"The article on speech synthesis is very good." A.R.B., Mexico City

"I liked 'The Architecture of a Capability-Based Microprocessor System'...more on data communications systems." B.G.Y., Inchon, Korea

"I liked 'The Architecture of a Capability-Based Microprocessor System.'" G.P.M., Warsaw, Poland

"I liked 'The Architecture of a Capability-Based Microprocessor System.'" A.R.V., Bruyeres-le-Chatel, France

"I disliked the articles of a research-oriented nature. Let's see more applications." R.T., Newark, DE (More applications-oriented articles are planned.—J.F.)

"I need to obtain more information on products (catalog, price, etc.)." T.V.N., Los Alamitos, CA (Unlike some commercial trade magazines, contributed technical articles are not followed by a "bingo" number for more information. Please contact the author directly.—J.F.)

"Dr. El-Imam's paper has an error on page 8. (D/A converter should be A/D converter)...." C.L.B., Arlington, VA (You are correct. Several other readers noted the error also.—J.F.)

"I liked MicroLaw...." R.P.B., Phoenix, AZ

"Please print name AND date on the bottom line together on EACH page, e.g., IEEE MICRO/August 1987 – Be proud of your work! Show it!" R.L.Z, Munich, West Germany (Excellent suggestion. I will pass this on to the Publications Board and the Publisher with my full endorsement.—J.F.)

"I liked all the articles." A.P, La Plata, Argentina

"I would like to see more articles of the quality of Yousif A. El-Imam. This one was excellent as it informs but also teaches." A.R.B, Mexico City (I have received several very favorable comments about this article. Author El-Imam did an excellent job.)

"I liked elegant design in capabilities article; MicroLaw desperately important." D.E., Perth, Australia (To the best of my knowledge, the MicroLaw-type legal column is unique to *IEEE Micro*.—J.F.)

More about the cover

Complete systems fabricated in gallium arsenide are possible in the very near future. Such systems can be very valuable because they function well in extremely hot or cold environments.

Researchers involved in one digital signal processing project found that the fabrication of a GaAs microprocessor would ease the integration of complete systems. After considering a number of microprocessor architectures, they selected the RISC, or reduced instruction set computer, as the best approach. This research, funded by the US Defense Advanced Research Projects Agency known as DARPA, ultimately produced single-chip designs based on two different GaAs technologies.

On the right of our cover you can see a microphotograph of a 32-bit Data Path demonstration chip, detailing 76 percent of the complete CPU chip. Our thanks to McDonnell Douglas Astronautics Company for supplying this picture.

The microphotograph appearing on the left side of our cover depicts a complete prototype CPU chip from Texas Instruments Incorporated. *IEEE Micro* also extends its thanks to TI and to the Mayo Foundation authors who prepared the article, "A 32-Bit, 200-MHz GaAs RISC for High-Throughput Signal Processing Environments." This article, which appears on page 8 of this issue, relates the background, current status, and fabrication expectations of the GaAs projects.

MicroReview

Richard Mateosian
2919 Forest Avenue
Berkeley, CA 94705
(415) 540-7745

Producing technical papers

Publication is the medium by which we communicate the results of our technical work to others. We do this to advance our chosen fields and to obtain the recognition of our peers. Sometimes this publication is an article in a professional journal like *IEEE Micro*; sometimes it's an internal document for a company or governmental agency; sometimes it's a set of lecture notes. Technical publication takes many forms, but they all have this in common: The greater the clarity and the better the appearance of the document, the greater the chance of its achieving the desired result.

A journal like *IEEE Micro* takes much of the responsibility for the appearance of articles appearing in it, but for conference proceedings and other similar publications, authors are often responsible for providing "camera-ready" copy, which is simply reproduced unchanged in the final book. Furthermore, in this era of desktop publishing, standards have risen for all forms of communication. Readers have come to expect reasonably well-designed and clearly printed memos, notes, manuals, and so forth. They are not as willing as they once were to plow through scrawled notes or poorly typed and badly reproduced papers. If you want to build and keep an audience for your work, you can't just let the content speak for itself; you must turn your attention to the quality of the presentation.

Of course, the content of your work is still the foundation of your technical reputation. Thus, the ideal tool for producing your work will allow you to achieve a high quality of presentation without forcing you to become a "wizard" for the tool itself. This month I've looked at two programs for the Macintosh that are capable of handling the kinds of papers that *IEEE Micro* readers are most likely to be called upon to produce. Although both of these tools need to be learned and provide ample opportunity for wizardry, they are capable of producing high-quality output for the casual user. One uses the popular "What You See Is What You Get" (WYSIWYG) approach, while the other is a formatter, i.e., it processes input consisting of text interspersed with formatting commands.

The trade-offs between these two approaches are well known. WYSIWYG is easy to use and fast, especially for small jobs, but what you see on your screen may not be what you get on your laser printer or typesetting machine. Formatting commands give precise control, but you don't know what you're going to get until you see the printed output. Also, formatting systems tend to be slower than WYSIWYG systems.

The following discussion is based upon my use of these products on a Macintosh Plus with a Data Frame XP20 hard disk, interfaced through the SCSI port.

Word 3.01, Microsoft (Redmond, Wash., 1987, $395.00)

Word 3.01 is a pretty impressive WYSIWYG word processor. It has many features, but you have to hunt for them. Word 3.01 comes with a 150-page tutorial book, which only hits the highlights. After that you have to dig everything out of the 458-page manual, which is arranged like an encyclopedia—alphabetically by topic with no logical structure and no introductory or tutorial material. The on-line help facility is extensive and convenient, but you have to know what you're trying to do before you can use it.

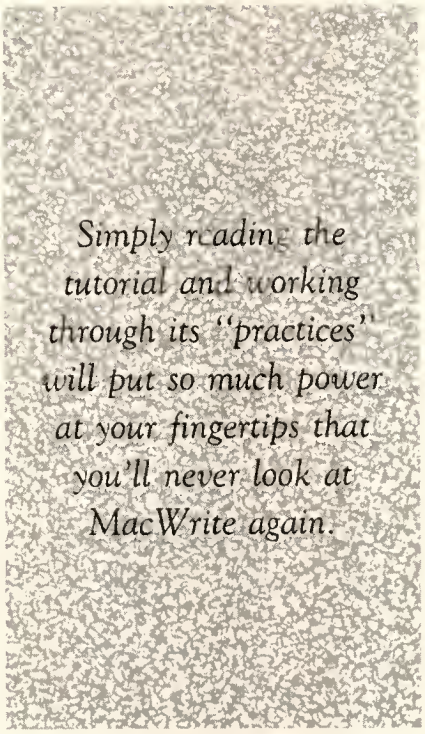
Simply reading the tutorial book and working through the "practices" in it will put so much power at your fingertips that you'll never look at MacWrite again. Then you can browse in the manual at your leisure or turn to it when a specific problem needs to be solved. That's the stage I've reached, and I've been pleased by the readability of the

manual and the ease of finding an appropriate feature for any problem I've had to solve. The only thing I've looked for that I haven't found is a macro facility. I didn't have a specific problem that I was trying to solve with macros, so I can't say whether an alternative was available.

Word lets you do just about anything that you need to do to produce a paper. You can start by outlining the material and then transform the outline into a paper, always keeping the outline structure available as a hidden part of the document. You can assign names to "styles," which are collections of paragraph and character formats, thus achieving uniformity of appearance within your paper or between papers. Footnoting, subscripting, superscripting, creating page headers and footers, page numbering, date and time stamping—all are easily accomplished with Word. You can format tabular material easily and even perform simple computations on selections of table entries. Glossaries allow named blocks of text and graphics to be placed in your paper with a few quick actions. A standard dictionary, which you can augment with your own special terms, supports spelling checking. You can prepare graphics using any Macintosh drawing or painting program (or charts using Excel), switch between these programs and Word with Apple's Switcher (supplied with Word), and place these exhibits in your paper.

You can easily resize and manipulate graphics with Word. If you're a Postscript expert, you can even include Postscript commands in your paper, allowing you to control the laser printer directly.

One area in which Word deviates from pure WYSIWYG and adopts the formatter approach is the production of formulas. You can type simple subscripts and superscripts, sums and products, and equations directly, but Word employs a functional notation for more complex forms (e.g., fractions, integrals, indexed sums and products, roots, matrices). Word has a general facility for toggling between two screen display modes: the WYSIWYG display and a display that shows visible representations of spaces, tabs, carriage returns, and other such characters. The functional notation is entered in the latter of these modes, and toggling to the former causes the specified formula to be displayed in final form. Nothing could be more convenient.



I've focused here on features relevant to producing technical papers and simple manuals. I haven't mentioned multicolumn printing, form letters, automatic hyphenation, page previewing, sorting, searching, interchange formats, and much more. There is even an entire set of keyboard commands that allows "mouse-free" operation for those already familiar with Microsoft Word for the MS-DOS environment. Microsoft's stated goal of keeping its MS-DOS and Macintosh products "in sync" is another plus for Word.

This unrestrained praise of Word should not be taken as a recommendation of Word over any competing product. This is not a comparison study. Other word processors may provide similar functionality or a better price/performance ratio. Simply take this as a baseline: Given what Word offers, there is no need to put up with less.

TeXtures, Addison-Wesley, (Reading, Mass., 1987, $495.00)

We have to begin with a discussion of etymology and pronunciation. TeXtures is spelled with an uppercase X in the middle because its name is derived from Donald Knuth's TeX. TeX is an approximation used on ordinary keyboards for TEX. This is the uppercase form of the Greek word spelled *tau, epsilon, chi,* which means art and technology. Thus, says Knuth, TeX is pronounced to rhyme with *blecchhh*, and all words formed from TeX share that pronunciation. One who masters TeX is a TeXnician, not a TeXpert, and the name of the product being described here is pronounced *tecchhhtures*, not textures. TeXtures is named as a contraction of "TeX plus pictures," because it marries the powerful formatting of TeX with the graphics capabilities of a Macintosh.

TeX is an edifice far too grand to be described adequately here, but I'll try to talk about some of its simpler features. TeXtures is an implementation of TeX for the Macintosh.

Every implementation of TeX has as its core the formatter described in *The TeXbook*, the first of Knuth's five-volume set called *Computers and Typesetting*. Thus, *The TeXbook*, a work of 483 pages, is included with TeXtures and forms the major part of the documentation. The remainder of the documentation is the 123-page user's guide, which describes the part of the package that is specific to this implementation of TeX. Thus, the user's guide deals with a small built-in editor, the facility for handling graphics, and the mechanism for previewing the final output on the Macintosh screen.

The editor is primitive, but adequate. Its purpose is rapid correction of errors uncovered during the formatting process. The file of text and formatting commands supplied to TeX is normally created with a standard editor, then imported by TeX.

Graphics are handled with the "picturebook," which works like a Macintosh scrapbook. You can copy graphics from other Macintosh application programs and then transfer them from the clipboard to the picturebook, where they can no longer be manipulated. Graphics are stored by name in the picturebook, along with precise size information for placing the graphic in the document. You can easily specify scaling to any arbitrary proportion of actual size.

You can preview the final output on the Macintosh screen one page at a time at any magnification from 10 percent to 500 percent of actual size. A magnifi-

Continued on p. 89

MicroNews

MicroNews features information of interest to professionals in the microcomputer/microprocessor industry. Send information for inclusion in MicroNews ***one month before cover date to Managing Editor,*** IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

"We don't need hype," Robert Hecht-Nielsen

Neurocomputing: a new information processing paradigm

Christine Miller, Assistant Editor

Whaaat? They're building a brain? A machine that sees, talks, and thinks? Are we ready for this?

IEEE Micro wondered and decided to approach Robert Hecht-Nielsen of the Hecht-Nielsen Neurocomputer Corp. We wanted to find out exactly what's happening in neural network technology and what all this means to the microcomputer industry.

Neural networks model the way the brain encodes and processes information. They recognize human faces, teach themselves to read and speak, learn from experience, and perform a variety of other pattern-recognition tasks that have baffled conventional computers.

Hecht-Nielsen is one of the first of many to develop neural networks for commercial applications and has built a neurocomputer. We plan to present other approaches in future issues.

Observers have called neural networks an emerging field. Do you agree?

Yes. Neural networks is one of the fastest growing areas of information processing, if not *the* fastest growing area. As more people see the capability demonstrations, they become aware that neurocomputing is a totally new information processing paradigm with numerous applications.

Since neurocomputers are not programmed, there is no need for software or software development. Development time and cost is lower, and networks can be implemented in simple hardware, with parallel hardware. All in all, implementation cost of neural network applications is much lower than costs for developing other applications.

Another reason neural networks are growing is that they are easy to learn for almost anyone already in information processing. I predict that neural networks will become the approach of choice for use in problem areas where neurocomputing is the application.

Currently, neurocomputing is a weak technology because we only know how to apply it to a rather narrow range of problems. However, this range is expected to broaden as we learn more.

To your knowledge, how many companies are presently developing and/or marketing commercial applications for neural networks?

I'm aware of at least 60 or 70 companies developing applications. Industries involved in this research are defense, financial, entertainment, telecommunications, electronics, and aerospace, to name some.

DARPA [the Department of Defense Advanced Research Projects Agency] through Dr. Ira Skurnick is very active in researching radar, sonar, weapons, and novel sensing applications.

The financial industry is using neural network experimentation with high-dimensional analysis of data. For instance, they are using neural networks to assess problems with credit line usage, or trouble signs on loan applications, or to score credit applications.

Neural networks can often be applied where there is an enormous database to be analyzed, like those involved in forecasting the success of a new movie or a new product. Other areas of application include special effects in films, automotive manufacturing, airline fare management, securities, and robotics.

What does all this mean to the microcomputer industry?

Microcomputers are the ideal host platforms for neurocomputing and will be used with neural networks because of their flexibility. For instance, IBM PC ATs and compatibles can handle a wide variety of hardware and software attachments and can help provide numerous applications in a cost-effective manner.

Microprocessors are the universal path for applying neural networks across a broad base. Other ways are prohibitive in cost.

What approach are you using to implement neural networks?

The neurocomputer is a coprocessor board that plugs into the host PC, which controls the neurocomputer. The host PC is loaded with a User Interface Subroutine Library. Using the UISL, users can call neural networks from software running on the host microcomputer as if they were subroutines. This makes neural networks easy to use anywhere they might be appropriate in a particular information processing application. This coprocessor approach allows users to freely mix programmed computing and neurocomputing so that each can carry out the processing it does best. Actually, it's simple to run.

Also involved is Axon, the machine-independent language for describing neural networks, existing or new. Just as an algorithm can be expressed in software, neural networks can be described in neurosoftware languages such as Axon. This approach allows the design of a combined programmed computing/neurocomputing system to be documented and maintained in a manner similar to current software maintenance procedures.

We have created network packages—most customers are only interested in four or five main networks—to be loaded into the neurocomputer on disk. The package usually provides only a general description of the network; users can tailor it to fit their needs.

On what theories are your implementations based?

Our Anza product is not specifically based on any one of the theories, but will work with all neural networks: Rumelhart backpropagation, Hopfield, and Grossberg counterpropagation, and other networks by Grossberg and Kohonen. One package, for instance, uses the Spatiotemporal Pattern Recognition Network by Grossberg.

[*Rumelhart, Hopfield, Grossberg, and Kohonen, among others, pioneered the development of neural networks.—Ed.*]

What is your source of funding?

We have had two rounds of financing: (a) the seed round with some 15 private investors; and (b) syndicate venture capital company investment. I am not at liberty to divulge the amounts involved.

"It is irresponsible and extremely grandiose to think that anything can function like a brain. That is a long way off."

More about applications. How are they selected?

Our company offers courses to train individuals in neural network capabilities and limitations. These "domain experts" then go out into their areas of expertise, whether it be commercial or governmental, and identify possible applications. We will also assist in this process.

Speaking of domain experts, is there a potential for use with artificial intelligence and/or expert systems?

A great potential. These are very compatible technologies; they don't compete. AI people are applying neural computing alongside knowledge engineering technology. I think this area will be very fruitful over the next two years or so.

A group known as the "Connectionists" believe that computers will act like brains when they are built like brains. What is your comment?

Connectionists are a part of the neural network community who are trying to make a point: If you want behaviors more like humans or animals, you have to use a different paradigm than before. But it is irresponsible and extremely grandiose to think that anything can function like a brain. That is a long way off.

We're not talking about brain capabilities. AI has had its hype—machines that can see, hear, whatever, have been promised, but never delivered. Our present capabilities are modest in comparison, but still impressive. We don't need hype.

Will the new 32-bit operating systems affect the development of neural networks?

We're excited. Hecht-Nielsen will be fully compatible. The software on a microprocessor is limited now in terms of memory and pointer limitation. The universal applicability of microcomputers will be assured by unlimited address space and other features.

What else is new on the horizon for you?

We are now beta-testing a new package called the Adaptive Resonance Network for hypothesis testing developed by Grossberg and Carpenter. As an illustration of how it works, consider an Automatic Teller Machine that can speak and listen to a client. The problem has been getting the software to focus on one character at a time. With this package, the teller machine focuses on the voice of the person as distinct from background noise, or in sonar it can focus on just one vehicle. We expect this package to be available the first quarter of next year.

IEEE Expert Call for Papers

IEEE Expert invites articles on **AI Applications** including **MIS/Financial Systems, Tools and Techniques, Health Care, Engineering/Manufacturing,** and **Education**—on **Technology** including **Real-Time Systems, Databases, Vision/Robotics, Software Engineering, Search, Natural Languages,** and **Knowledge Engineering**—plus **Book Reviews, Conference Coverage, Short Subjects,** and papers on **PCs, Products,** and **Resources.** Please submit articles to Editor-in-Chief David Pessel, BP America, 4440 Warrensville Center Rd., Cleveland, OH 44128. Please submit other papers to Henry Ayling, Managing Editor *IEEE Expert,* 10662 Los Vaqueros Circle, Los Alamitos, CA 90720. Book reviewers please contact Associate Editor K.S. Shankar, Federal Systems Division, IBM Corporation, 3700 Bay Area Boulevard, Houston, TX 77058.

Continued on p. 90

FEATURE

A 32-Bit, 200-MHz GaAs RISC

for High-Throughput Signal Processing Environments

Barbara A. Naused and Barry K. Gilbert
Mayo Foundation

GaAs technology has matured sufficiently to allow fabrication of an entire RISC on one chip. GaAs also supports 200-MHz clock rates and 100-MIPS instruction rates.

The first complex single-component microprocessor fabricated in gallium arsenide (GaAs) is now being developed as a major core element in a project known as the Advanced Onboard Signal Processor. Funded jointly by the Defense Advanced Research Projects Agency (DARPA) and the US Air Force, the AOSP will be assembled by 1990 entirely with GaAs components as a demonstration that digital GaAs technology will be sufficiently mature to be used in signal processing systems.

In the early 1980's, DARPA asked the Special Purpose Processor Development Group of the Mayo Foundation to identify a microprocessor architecture suitable for implementation as a custom-designed monolithic GaAs IC by the late 1980's. A number of commercial, aerospace, and Department of Defense microprocessor architectures were reviewed for their suitability. Here, we compare their strengths and deficiencies in the AOSP application context and discuss the selection of a RISC architecture for GaAs implementation.

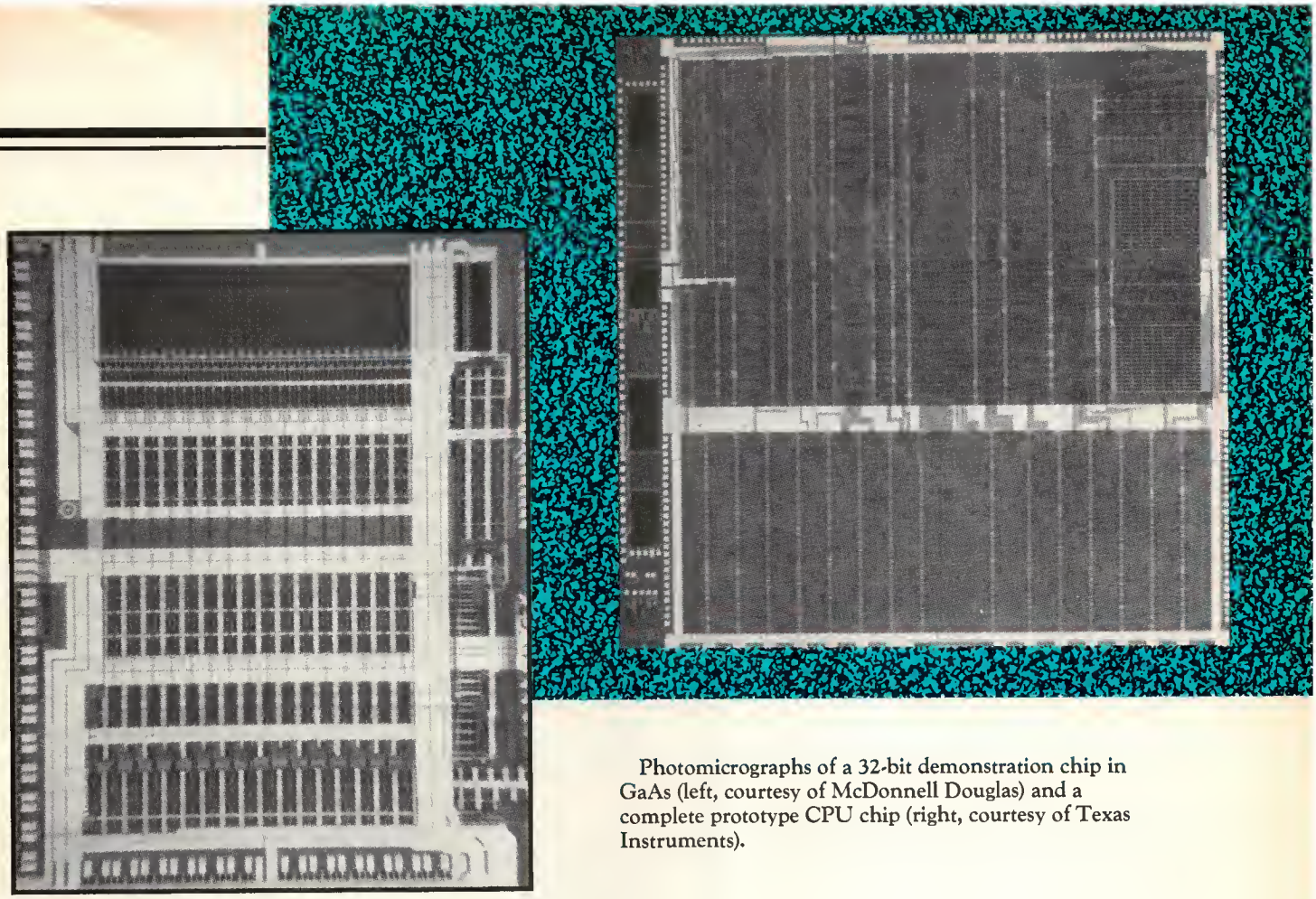
The AOSP

The AOSP is a general-purpose, distributed signal processor presently under development in silicon technology by Raytheon Corporation for use in several DoD signal processing applications. The architecture of the AOSP is based on a network of processing elements, called array computing elements (ACEs), which can all be identical, or can be a variety of types to perform several different functions. A distributed operating system controls the network, which is specialized for signal processing. The ACEs communicate through an efficient interconnection network which also allows spare ACEs to be substituted for faulty ones in real time (Figure 1). Each ACE contains two sections:

- *the Network Control Unit* (NCU), which is responsible for communication with the network and control of the ACE, and
- *the Application Processing Unit* (APU), which is optimized for execution of signal processing applications and also interfaces directly with the outside world.

Each NCU consists of an interelement bus interface and a control processor. The APU contains a

0272-1732/87/1000-0008$01.00 © 1987 IEEE



Photomicrographs of a 32-bit demonstration chip in GaAs (left, courtesy of McDonnell Douglas) and a complete prototype CPU chip (right, courtesy of Texas Instruments).

signal processor, referred to as the Macro Function Signal Processor (MFSP), a system I/O unit, and a data processor for scalar operations. A Motorola 68010 microprocessor functions as the control processor in the NCU and the data processor in the APU, as illustrated in Figure 2.

As the design of the AOSP progressed in silicon, DARPA management realized that certain applications would require very high clock rates, extreme radiation hardening, and/or very low power dissipation and that a GaAs implementation of the AOSP might achieve these combined features.[1]

Several studies conducted during 1981 and 1982 assessed the level of technology required to achieve a radiation-hardened, low-power signal processor by the end of the decade. These studies used the AOSP architecture as a baseline. Results indicated that GaAs IC technology would have to be sufficiently sophisticated to permit fabrication of configurable cell or gate arrays in the size range of 4000 to 6000 equivalent gates, as well as static RAMs (SRAMs) of at least 16K bits.

Assessments of the AOSP architecture indicated that—with one exception—an entire AOSP ACE could be fabricated using only the gate arrays and SRAMs. To assure that these technologies would be in place by the mid-1980's, DARPA embarked upon an ambitious plan to fund the creation of three pilot fabrication line facilities for GaAs chips. The products of the first two pilot lines were to be gate arrays of at least 6000-equivalent-gate complexity, and SRAMs of 16K-bit complexity. These projects are doing well. A Rockwell/Honeywell team demonstrated 5500 gate arrays in late 1985, and both Rockwell and McDonnell Douglas demonstrated 4K-bit SRAMs in early 1986. Rockwell demonstrated fully functional 7000 gate arrays in mid-1987 and fully functional 16K-bit SRAMs in September 1987.

A single exception to the assembly of an entire AOSP ACE based upon GaAs gate arrays and SRAMs remained: the requirement for a microprocessor to serve as the control and data processor elements of the AOSP ACE, as we earlier discussed. In order to achieve the full benefits of digital GaAs technology, the entire AOSP ACE had to be fabricated with GaAs components, including the microprocessors. However, further studies indicated that no modern microprocessor could be assembled efficiently with gate arrays. A custom IC was necessary to achieve optimum performance. Because GaAs IC technology was then, and remains, relatively immature in comparison to silicon technology, the device densities of GaAs gate arrays and custom components are considerably less than in silicon. As a result, a major constraint was placed on the design of a GaAs microprocessor. It would have to require less than 10,000 equivalent gates to allow its placement on a single custom chip.

The AOSP application placed additional constraints on the design of a microprocessor:

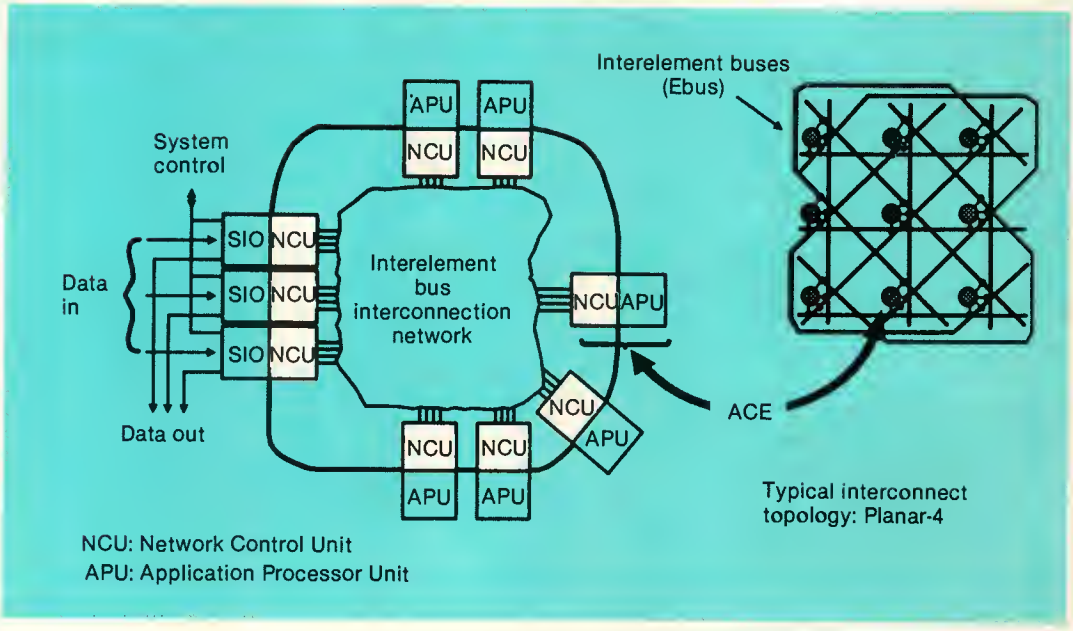


Figure 1. The AOSP system architecture developed by Raytheon Corp.

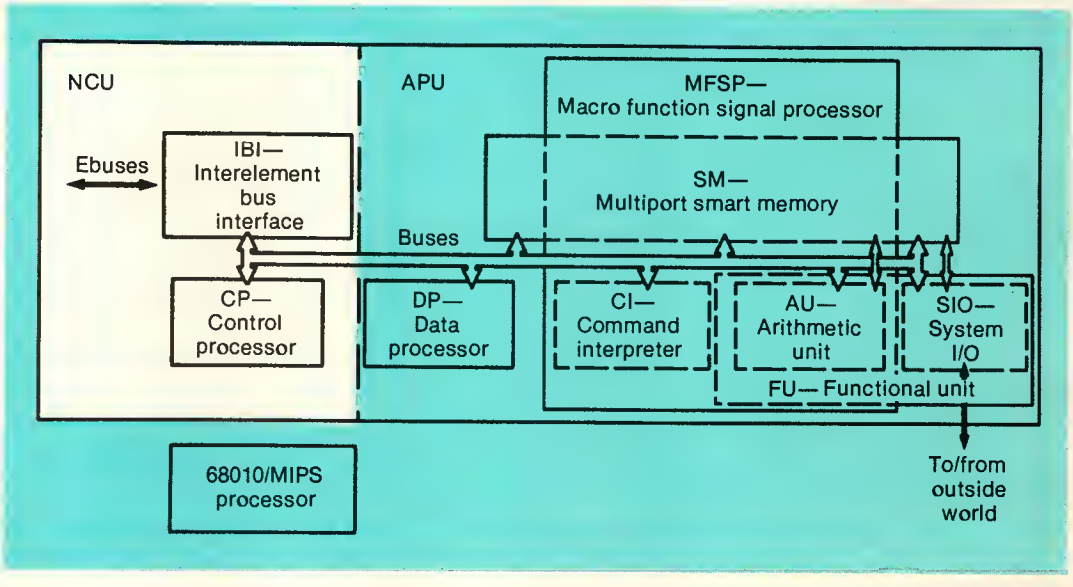


Figure 2. AOSP Array Computing Element (ACE) showing both the Network Control Unit (NCU) and the Application Processor Unit (APU).

- a 32-bit architecture;
- a 24- to 32-bit address bus to support at least 16M bytes of direct memory access (DMA);
- an efficient I/O handler;
- an efficient interrupt handler;
- a high-instruction execution rate; and
- the capability of floating-point operations in hardware.

If necessary, the floating-point hardware could be a separate coprocessor chip. Accordingly, DARPA tasked us to identify a 32-bit microprocessor architecture for full implementation in GaAs in time to achieve an all-GaAs AOSP brassboard demonstration in 1990.

Microprocessor architectures

Although a microprocessor could have been designed specifically for GaAs implementation, we did not consider this type of approach attractive without guaranteed software support. The possible use of an existing architecture therefore added another constraint to the GaAs microprocessor selection process. A microprocessor design that was nonproprietary, readily available, and well-documented was required. These constraints formed the basis of our study.

Commercial architectures. We first investigated commercially available microprocessor architec-

Table 1.
Possible architectures for use in a GaAs microprocessor.

	Gate count < 10K	32-bit architecture	Non-proprietary	DMA ≥ 16M bytes	Execution > 1 MIPS
Standard architectures					
Mil-Std 1750A (TI)			●		●
TI 9900					
MC68010 (12 MHz)		●		●	●
NS16032		●		●	
Nebula		●	●		
Intel iAPX432		●		●	
Rockwell AAMP				●	
RISC architectures					
RISC II - Berkeley		●	●	●	
IBM 801	?	●		●	●
Inmos transputer		●			●
MIPS - Stanford	●	●	●	●	●

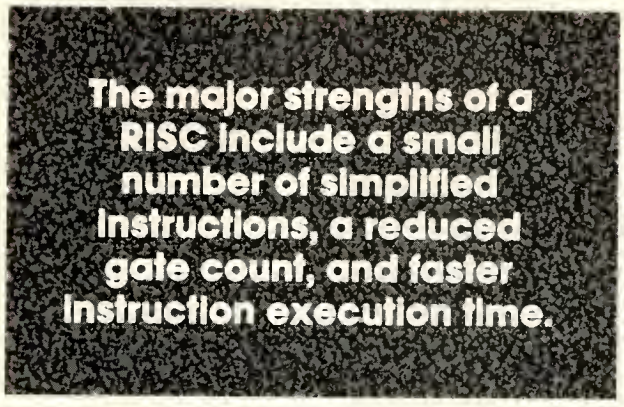
tures—in particular, the Motorola 68010. Because this microprocessor is used in the silicon version of the AOSP under construction by Raytheon, it would have provided a straightforward target architecture. However, the gate count of this processor is too large for a single-chip implementation in GaAs at projected late-1980's integration levels. Further, the design for the 68010 is proprietary and would not be made available. Raytheon discovered additional drawbacks of the 68010. Wait states are incorporated in the initialization of I/O operations, and the data bus is only 16 bits wide. Both of these features are inefficient in the context of the AOSP. We did not consider it a drawback that the 68010 requires a coprocessor for floating-point operations, provided that the coprocessor could be made to operate at sufficiently high speed.

We also examined and rejected the following commercial and aerospace industry architectures on the basis of one or more of the criteria previously described: the National Semiconductor 16032, the Texas Instruments 9900 series, the Rockwell International AAMP (Advanced Architecture Microprocessor), and the Intel iAPX 432 (Table 1). These microprocessors shared a principal drawback—their large gate counts.

Military architectures. We also considered a GaAs implementation of the Mil-Std-1750A[2] architecture because of the large amount of software written for this processor. Because this design is a DoD standard, there was considerable initial enthusiasm within the DARPA community for its selection as the target architecture for the GaAs microprocessor project. However, Mil-Std-1750A specifies a 16-bit architecture with a single 16-bit bus, yielding only 128K bytes of DMA, half of which is used for instructions and half for data. A memory management unit can be appended to the processor in most versions to expand the DMA capability to 2M bytes. However, this approach is not efficient for the large number of array-oriented and I/O operations required in the AOSP. Most implementations do support on-chip floating-point operations and also contain an efficient interrupt handler. Sixteen general-purpose, 16-bit registers are also provided. A very large number of instructions and addressing types are available in this architecture, yielding a variety of instruction formats and lengths. Control logic for the 1750A is implemented with microcode. One implementation of Mil-Std-1750A by Texas Instruments[3] requires two chips, each containing over 16,000 gates and 110K bits of microcode ROM.

We rapidly concluded that Mil-Std-1750A could not provide a suitable GaAs microprocessor architecture for the AOSP project. The processor architecture is too complex and yet does not implement a 32-bit machine. Computation with 64-bit operands is not supported, and the direct address space is considered too small, given the trends toward ever-increasing direct memory capacity.

We also examined the Nebula machine (Mil-Std-1862B), a DoD 32-bit architecture, for implementation in GaAs (Table 1). However, this architecture is not necessarily a single-chip microprocessor, even in silicon VLSI. One of the multichip implementations



attempted to date by Raytheon Corporation requires approximately 44,000 gates, which is much too large for a GaAs version, at least through the end of the decade. The instruction execution rate of this machine is also comparatively slow at 500 KIPS. Therefore, we also eliminated this architecture from further consideration.

RISC architectures. The family of reduced instruction set computers (RISCs) appeared to match the constraints of a GaAs microprocessor more closely than the machines described previously. This type of computer, developed independently by several universities and corporations, was considered a novel approach in the early 1980's. Only recently has it become commercially available in silicon. The family of RISC architectures was developed after it was recognized[4] that a large number of complex instructions normally implemented in hardware on a microprocessor are rarely used in compiled or manually prepared assembly-language code. These infrequently used instructions could therefore be more efficiently implemented in software, reducing the hardware support to a small set of simple instructions.

This design approach can reduce the gate count of the processor considerably, especially in the control section, and can permit the remaining simple instructions to execute much more rapidly. The simplified hardware instruction set complicates the low-level software required for the microprocessor, but this price is paid only once at compile time, rather than every time an instruction is executed. Compile-time penalties are not a major concern if overall performance is improved and the simplified microprocessor exhibits a considerably reduced gate count. Several versions of RISCs had been introduced at the time we performed this architecture study, including the RISC II processor design by the University of California at Berkeley, the 801 minicomputer design by IBM, the transputer design by Inmos Ltd., and the MIPS (Microprocessor without Interlocked Pipeline Stages) design by Stanford University (Table 1). As we will discuss, this last architecture appears to be best suited for implementation in GaAs.

The major strengths of a RISC include a small number of simplified instructions, a reduced gate count, and faster instruction execution time. Several other architectural aspects naturally lend themselves to straightforward hardware implementation on this type of machine (although not every RISC employs all of these features). These features include:

- a load/store architecture (only load and store instructions access memory, while all others operate on registers);
- single-cycle execution of most instructions;
- a short critical path;
- a hardwired control section (rather than microcode);
- a Harvard architecture (separate memories and buses for data and instructions);
- a fixed-instruction format (all instructions are the same size and structure);
- preprocessing of pipeline interlocks in software; and
- the ability to keep major resources of the machine (the ALU, the data memory, and the instruction memory) fully occupied most of the time.

The RISC II. The RISC II[4] contains a large on-chip register file used for storing instructions, local variables, and constants. The register file consists of eight overlapping windows of 32 words each (a powerful support structure for procedure calls). However, the file must be loaded one word at a time. When this machine is interrupted, the contents of the entire register file must be stored in off-chip memory—a very time-consuming operation. The machine supports only 31 instructions in hardware, each of which is one word (32 bits) in length. All instructions execute in one clock cycle, except load or store operations, which require two cycles. The RISC II contains a three-stage pipeline. The compiler targeted for this machine rearranges the instructions to allow useful operations to be executed during pipeline conflicts. The RISC II designers intended that users would almost always program in high-level languages (HLLs), thereby allowing this associated compiler to optimize the code before execution.

Because only a small portion of the chip area is devoted to control, the floor plan of the device is very regular. The addresses and data words are multiplexed to reduce the total I/O count for the chip to 50 pins, but the multiplexing degrades I/O operation speed. Programs written for the RISC II are approximately 50 percent larger, but typically run faster, than programs written for the more conventional machines we described earlier. Floating-point operations are handled off chip, and integer multiply and divide operations are executed in software. Interrupts are supported only with an external interrupt flag; all other interrupt operations must be processed off chip. The inability of the RISC II to process interrupts on chip is a significant limitation in the AOSP application environment.

The complexity of the RISC II is 12,000 gates, which could have been reduced somewhat in a GaAs version by decreasing the size of the register file at the cost of a severe performance degradation. Although the RISC II offered many attractive features, it did not appear to be efficient for the type of application typical of the AOSP environment, and therefore we did not recommend it for implementation in GaAs.

The IBM 801 machine. The 801 minicomputer designed by IBM[5] is a RISC in which the most recently used instructions are stored in an instruction cache which has an access time of one machine cycle. A data cache is also provided. Data from the two caches are fetched asynchronously to one another, and it is possible to overlap access to the two caches. A software I/O manager synchronizes the caches when required, minimizing the execution of unnecessary load and store operations. A compiler reorders the code to allow useful instructions to be executed during branch and load delays. This processor was also intended to be an HLL-computer, using a very efficient compiler to produce object code nearly as efficiently as the best hand-generated code.

Thirty-two general-purpose, globally allocated registers are available in the 801 machine, so that data needed again within a short duration is available immediately. Instructions are 32 bits in length and have been optimized for microcode. Complex instructions are executed in software. Addresses and data values are also 32 bits in length. A fixed-precision multiplication can be completed in 16 cycles, while a division can be completed in 32 cycles. Floating-point operations are executed off chip with the 801. This machine averages 1.1 cycles per instruction, exhibits cache hit ratios close to 100 percent, and contains a two-stage pipeline. The number of instructions required for a program varies considerably. In some cases, instruction streams are equivalent in length to those found in a more complex processor. In other cases, a 50 percent increase in the number of lines of code is required, particularly for applications requiring many floating-point operations. The information to determine if this machine has an acceptable gate count for a GaAs implementation was not available. The architecture is proprietary to IBM, and additional information is not accessible by the public. This machine appears to match the AOSP environment and might be applicable to a variety of tasks in GaAs if the design were made available.

The Inmos transputer. The transputer is a very high speed 32-bit microprocessor executing a reduced-instruction set of 59 instructions.[6] The current implementation requires approximately 62,500 gates, including 4K bytes of SRAM, a memory interface, a peripheral interface, and on-chip serial links to other transputers, as well as the main processor. A transputer can be used alone, although the intention as stated by Inmos is to employ networks of transputers to increase the performance of an entire system. The same software can be used regardless of the number of processors used. The transputer, which responds very quickly to external interrupts and supports simultaneous block transfers, was designed to implement concurrent processes. Instruction sequences are executed with no wait states.

The MIPS machine was the only architecture which satisfied all constraints of a GaAs microprocessor for the AOSP.

The transputer can be programmed in most high-level languages, but is intended to perform most efficiently when programmed in Occam, a language specifically developed by Inmos to exploit concurrency. Occam eliminates the need for assembly-language programming and is the lowest level language supported by the transputer. Floating-point operations are executed in software in the early versions of the transputer, and, we believe, are to be executed in hardware in the later versions. The transputer supports a single 32-bit bus, which must be multiplexed for data and addresses. The transputer employs an internal microengine to execute its assembly-level instructions, and possesses no completely general-purpose registers in the hardware. These features are unusual in a RISC. The transputer is also a proprietary architecture. If this design had been available, we might have considered it as a replacement for the entire NCU portion of an AOSP ACE (implemented with multiple chips), rather than merely as the control processor in the NCU.

The MIPS machine. Stanford University originally developed the MIPS machine[7-10] in 1981 as a RISC-architecture project under DARPA sponsorship. After examination, the MIPS machine was the only architecture which satisfied all constraints of a GaAs microprocessor for AOSP. The philosophy behind the MIPS project was to implement in hardware only the most frequently executed and time-consuming instructions and to implement infrequently used instructions in software. This implementation of the instructions increased the overall speed of operation of the processor. The MIPS machine architecture and supporting software were available for government-supported projects. The MIPS is a very high speed 32-bit architecture, primarily due to an efficient match between its architecture and its supporting software.

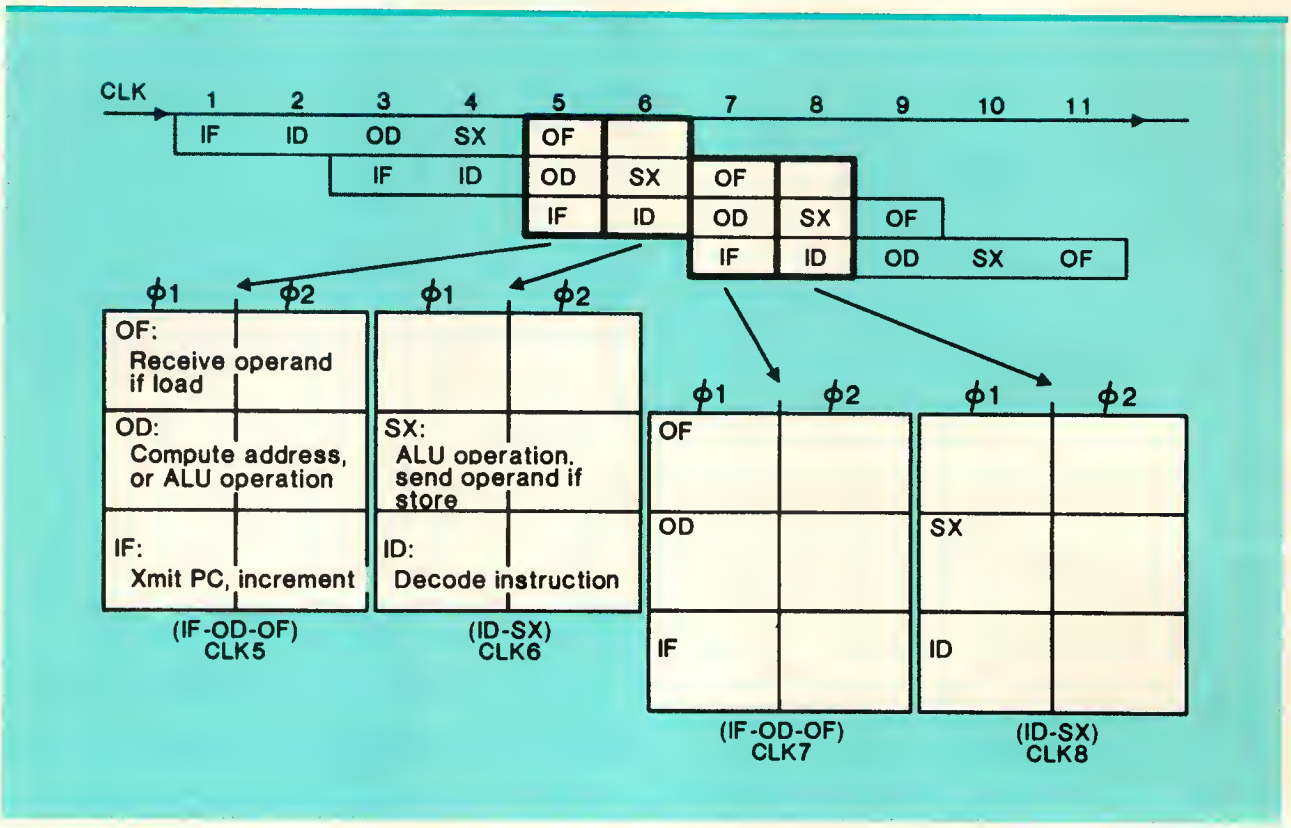


Figure 3. Stanford MIPS five-stage pipeline with active pipe states for each clock cycle. Reproduced in part from Hennessy et al.[9]

The Stanford MIPS machine contains a five-stage pipeline, with a new instruction entering the pipe every other cycle, yielding three active instructions per cycle. Each pipestage executes in the same amount of time, and every machine instruction passes through each stage. The five stages are called

- *IF* (instruction fetch), which transmits and increments the program counter for fetching the instruction;
- *ID* (instruction decode), which decodes the instruction;
- *OD* (operand decode), which either computes the memory address for a load or store instruction, computes the program counter for a branch instruction, or performs an arithmetic operation;
- *SX* (store and execute), which transmits the operand for a store instruction, and, in addition, either performs an arithmetic operation or performs the compare for a conditional instruction; and
- *OF* (operand fetch), which receives the operand for a load instruction.

The machine alternates between the IF-OD-OF cycle and the ID-SX cycle, as long as no interrupts occur, as shown in Figure 3. This pipeline implementation enables 100 percent utilization of the major hardware resources of the processor, which include the instruction memory (by the IF and ID pipestages), the ALU (by the OD and SX pipestages), and the data memory (by the OF and SX pipestages), as depicted in Figure 4. Because two pipestages are able to execute arithmetic operations, two adds can be packed into one machine instruction and executed within one machine cycle. A 32-bit fixed-precision multiply can thus be executed in eight cycles, with a 2-bit Booth's algorithm executed at 4 bits per cycle. An ALU operation can also be packed with a load into one machine instruction in this architecture.

Interlocks are necessary in pipelined microprocessor architectures to prevent instructions from interfering with one another as they traverse the pipeline. The MIPS machine implements its pipeline interlocks in software, rather than in hardware as is customary in other microprocessor designs. This implementation eliminates a significant amount of complex control logic from the processor. Interlock arbitration can be accomplished in software because the information necessary to generate the interlocks is known at assembly time. By moving this function from execution time to compile time, execution time becomes much faster. The MIPS uses a sophisticated reorganizing assembler which is tightly coupled with the simpler hardware. This "reorganizer" assembles the code into executable machine code, packs two independent instructions into one instruction where possible, and reorganizes the code. Code reorganization accomplishes pipeline interlocking by substituting useful instructions from elsewhere in the code stream where No-op instructions would otherwise have to be inserted to avoid pipeline conflicts, as shown in Table 2. Reordering of instructions improves code execution speed an average of 20 per-

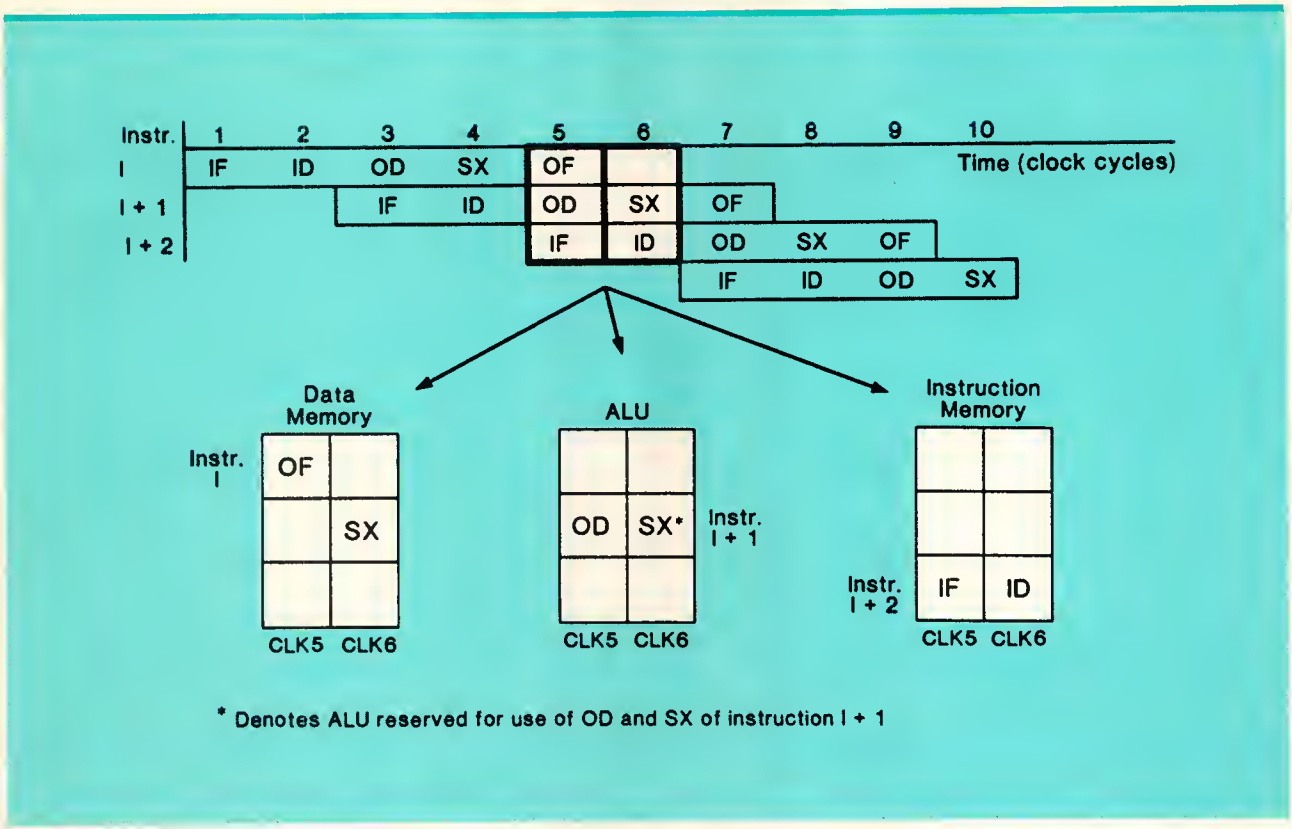


Figure 4. Stanford MIPS five-stage pipeline showing hardware resource utilization by each pipestage. Reproduced from Przbylski et al.[7]

Table 2.
Example of MIPS code before and after reorganization and packing by the sophisticated assembler.*

	Source code	Correct code with No-ops	Reorganized code
(For Loop) →	Begin		
	A[i]:=B[i]+C[i];	L20: ld (r4, r1), r6	L100: ld (r4, r1), r6
		ld (r5, r1), r7	ld (r5, r1), r7; add r6, r8
		No-op	
		add r7, r6, r9	add r7, r6, r9; add r7, r10
		st r9, (r3, r1)	st r9, (r3, r1); add #1, r1
	R:=R+B[i];	add r6, r8	
	S=S+C[i];	add r7, r10	
		add #1, r1	
		ble r1, r2, L20	ble r1, r2, L100
		No-op	st r8, R(sp)
		No-op	st r10, S(sp)
	End	st r8, R(sp)	
		st r10, S(sp)	

*Note that the code length has been reduced by almost half. Reproduced from Hennessy et al.[10]

cent.[7] Instruction packing increases code density, and reduces the execution time of an instruction stream by an additional 30 percent. This gives a combined improvement in execution speed of over 50 percent, when compared to the execution speed of code produced by traditional assemblers. Because the reorganizer accepts MIPS assembly code as input and produces machine-executable code as output, this software module must be unique to each hardware implementation.

The Stanford MIPS machine implements conditional control flow with software by using a compare and branch instruction, which executes in a single machine cycle. The MIPS does not use condition codes—the typical method of implementing conditional control flow—because the structure of the hardware typically employed for condition codes is irregular, both in design and in physical layout. Thus, condition codes are an inefficient use of space on the chip. A total of 16 comparisons, both signed and unsigned, are available in the ALU for this purpose. The MIPS also implements a "set conditional" instruction with the same 16 comparisons.

The Stanford MIPS machine employs a Harvard architecture, having separate data and instruction memories which are accessed on alternating phases of the two-phase clock. Only load and store instructions access memory; all other instructions operate on registers (thus referred to as a load/store architecture). Five addressing modes are available: long-immediate, absolute, based, indexed, and base-shifted by n (where $0 < n < 5$). Although the Stanford MIPS is a word-addressed machine for both instructions and data, it has special instructions to support byte operations on data. Data is accessed with a 24-bit physical address, expandable to a 32-bit virtual address. This yields 16M words of directly addressable memory, with one level of optional page mapping off chip. In addition, if an instruction is *not* a load or store (approximately 50 percent of the instructions), a memory cycle may be used for cache write-back or prefetch.[7]

The Stanford architecture supports vectored interrupts with a 12-bit address concatenated to the current machine status for a jump to the proper interrupt routine. When any type of exception is detected, instructions that are currently in the pipeline are completed, if possible. The state of the processor is stored in the "surprise register." Execution is transferred to memory location zero, where the registers and three return addresses are saved for return to the code sequence that was in progress, and the interrupt handling routine is invoked.

The original MIPS chip was fabricated in NMOS and requires 8000 gates and 84 I/O pins. The chip layout is very regular, containing six major hardware sections. The control of the microprocessor is divided into two functional units implemented as PLAs: (a) the instruction decode unit, and (b) the master pipeline control. These occupy about 20 percent of the chip area. The data path is interconnected through a pair of 32-bit buses. It consists of the ALU, a barrel shifter, a register file (including 16 general-purpose registers), and the program counter and address mask unit. A control bus interfaces the data path to the control sections.

These chips have an average throughput rate of two million instructions per second and a power dissipation of 1.6W, at a clock rate of 4 MHz.[9] The Stanford MIPS was benchmarked against a Motorola 68000 and a VAX 11/780 by executing several computation-intensive programs written in Pascal. The results of these benchmarks demonstrated that the average throughput of the MIPS was five times faster than that of the Motorola 68000 and twice as fast as that of the VAX 11/780[8] (Table 3).

The MIPS instruction set originally defined at Stanford[7] contains 32 instructions. All of these instructions are 32 bits in length, have the same instruction format, and execute in a single machine cycle. The Stanford MIPS machine requires a coprocessor for high-speed execution of floating-point operations. However, if one is not attached, these instructions can be executed in software on the main processor.

Current projects in GaAs based on the MIPS architecture

After completing our 1983-84 studies, we felt that an entire MIPS-type microprocessor might be implemented as a custom-designed GaAs IC containing approximately 10,000 equivalent gates, because of the relatively low complexity of such a microprocessor. We also felt that a custom-designed floating-point coprocessor might be implemented in an additional 10,000 gates on a second chip.

Based on these results, in 1984 DARPA initiated a five-year program to develop a full 32-bit architecture microprocessor and floating-point coprocessor in gallium arsenide using the Stanford MIPS machine as the baseline. Three contractors—McDonnell Douglas, Texas Instruments teamed with Control Data Corporation, and RCA Corporation teamed with TriQuint Semiconductor—completed a one-year architecture study phase. A four-year, two-phase project to fabricate a 32-bit GaAs microprocessor is currently under way, both at McDonnell Douglas and at Texas Instruments. The goals for these GaAs microprocessors are

- a custom main processor chip and a custom floating-point coprocessor chip,
- a chip complexity of 10,000 gates,
- operation of both chips at a 200-MHz clock rate, and

Table 3.
Results of Pascal benchmarks in seconds.*

	MIPS	M68000	VAX 780 (appr.)	DEC 20/60
Clock speed	(4 MHz)	(8 MHz)	(5 MHz)	
Si transistor count	(25,000)	(65,000)		
Puzzle	2.40	6.1	5.2	2.6
Queen	0.44	1.9	1.0	0.5
Perm	0.56	3.0	1.2	0.6
Towers	0.64	2.9	1.4	0.7
Intmm	0.80	5.0	1.0	0.5
Bubble	0.58	3.7	1.4	0.7
Quick	0.41	2.6	0.8	0.4
Tree	1.01	9.9	2.0	1.0
Avg. (relative to MIPS)	1.00	5.1	2.0	1.0

*Reproduced in part from Przbylski et al.[7]

- a sustained throughput rate of at least 100 million instructions per second.

The first phase ended in March of 1987 with each of the companies producing

- the assembler, linker, and reorganizer software;
- stand-alone demonstration chips containing large portions of the microprocessor; and
- a more detailed system-level specification.

During the second phase, the actual chips and boards will be fabricated by the two contractors, and the systems will be integrated into single-board computers. DARPA plans to receive the first prototype GaAs RISC CPU chips and floating-point coprocessor chips by early 1988. DARPA then expects completion of the first GaAs single-board computers for the GaAs AOSP project in early 1989.

A computer operating at a 200-MHz clock rate presents a serious design problem: data starvation. The main memory currently cannot supply instructions and data operands to the processor at a sufficiently high rate to keep it occupied. The provision of fast cache memory helps alleviate this problem. Both GaAs RISC development projects plan to use two off-chip cache memories (one for instructions and one for operands) and two cache controllers or memory manager chips.[11,12] The caches will be necessary in both the autonomous single-board computer system and the GaAs AOSP. These caches will be 1K words each, with an optimum access time of one nanosecond and a worst case access time of 2.5 ns. The complete computer system will contain

- the RISC CPU,
- two floating-point coprocessors,
- two cache memories,
- two memory management units,
- a system controller for low-speed I/O and external exceptions, and
- main memory.

The CPU chip may also function as an embedded controller for a system which supplies its own memory and does not require the floating-point coprocessor.

We believe that there is also considerable growth potential in the GaAs version of a MIPS processor chip set, which could be explored as follow-ons to the present projects. For example, with more available real estate on the chips, a cache memory or additional registers could be placed on chip. The addition of on-chip cache would provide a significant performance gain, since in current GaAs RISC designs a transfer bottleneck occurs between the microprocessor and its off-chip caches. Alternately, some of the floating-point functions now performed on the coprocessor chip could be incorporated into the microprocessor itself.

In addition, Rockwell International preliminary studies indicate the possibility of a MIPS-based machine operating at a 500-MHz clock rate and executing more than 200 million instructions per second. This version of the MIPS machine would require second-generation GaAs transistors, such as HEMTs or HBTs.

Silicon versions of the DoD MIPS machine. The GaAs RISC project has had an impact on concepts within DARPA, the US Air Force, and the Strategic Defense Initiative Office regarding optimum

architectures for next-generation processors. In order to gain additional benefits from GaAs RISC developments in architecture, hardware, and software (see discussion on software), DARPA has initiated a parallel development in silicon. These microprocessors will be fabricated using bulk CMOS and CMOS/SOS to exploit the higher complexity levels available in silicon. Three contractors—Sperry, General Electric, and RCA—completed a one-year architecture study phase in 1986. Two of these contractors are now conducting a two-year phase to fabricate a silicon MIPS-type chip containing on-chip cache, and a floating-point coprocessor chip, with the target clock rate of 40 MHz. Silicon chip prototypes were successfully demonstrated by the two contractors in the late fall of 1987.

Establishment of a standard ISA and transportable software. The government agencies involved with these projects decided to establish a standard Instruction Set Architecture (ISA) for all MIPS-based machines. This standard provides software compatibility for all MIPS-based processors presently under development while still allowing flexibility at the hardware design level. An ISA is a list of attributes visible to an assembly-language programmer or to an HLL compiler. In general, an ISA description includes lists of data formats, instruction formats and mnemonics, addressing modes, available registers, memory and interrupt control structures, I/O operations, and detailed instruction-set requirements. However, an ISA does not include specific details for a given hardware implementation of a processor.

Carnegie Mellon University, Stanford University, Mayo Foundation, and the contractors currently implementing MIPS-based hardware collaborated to develop an ISA document for the DoD MIPS machines.[13] Their goal was to ensure that all versions of the MIPS processor will execute the same software on an interchangeable basis. This MIPS-based ISA currently serves as the baseline document for HLL compilers (in Pascal and Ada) under development through DoD sponsorship for the MIPS processors. Translators, or cross-assemblers, are also being written for the Motorola 68000 and for the Mil-Std-1750A microprocessors. Assembly-language programs previously written for these processors can then be used directly on any of the DoD-sponsored MIPS processors. This software is now available, with the exception of the Ada compiler. The Software Engineering Institute associated with Carnegie Mellon University is maintaining this software.

We now briefly describe several features required by the DoD-standard ISA for all current and future MIPS-based processors. The ISA increased the number of assembly-level instructions from 32 to 69. The additional instructions support

- unsigned arithmetic operations;
- byte, half-word, and word operations; and
- floating-point operations.

Most of these added instructions should still execute within a single machine cycle. Floating-point operations are executed in hardware on the coprocessor, or—in the absence of a coprocessor in applications not requiring high-speed floating-point operations—in software on the main processor. Single-precision integer and floating-point data operands are 32 bits wide, while double-precision data operands are 64 bits wide. At least 16 general-purpose, 32-bit registers and at least four 64-bit, floating-point registers must be available on these processors to satisfy the ISA. All data memory access is by explicit load and store instructions. Instructions continue to be addressed on word boundaries. However, addressing of data operands has been changed to a byte-addressed format. As a result of this change, data can be loaded and stored as byte, half-word, and word-sized operands. Only two addressing modes have been specified in this ISA: *absolute*, in which the address is specified directly; and *based*, in which the address is obtained by adding an offset to a base register.

Several changes not specified in the standard ISA have also been made, which appear to improve the efficiency of the GaAs implementations of MIPS in comparison to the original Stanford MIPS architecture. These changes include

- modification of the number of pipestages from five to three in one implementation and to six in the other;
- initiation of a new instruction every machine cycle, rather than every other cycle; and
- elimination of the capability of packing two assembly-language instructions into one machine instruction.

(Instruction packing unnecessarily complicates the design of the processor control section.)

An additional feature of the GaAs RISC implementations is that up to eight coprocessors can be attached to a single microprocessor chip. These coprocessors do not necessarily have to be floating-point coprocessors.

Coprocessor implementations. Because the coprocessor architectures currently being implemented with the GaAs RISC processors vary considerably, we present only one of the coprocessor implementations under development.[11] The microprocessor and its coprocessor operate synchronously from a common clock and have common instruction address, instruction data, and operand data buses. The microprocessor calculates memory addresses and initiates memory references and then continues executing its own instructions. The coprocessor accepts its own instructions and data and outputs its values either to mem-

GaAs RISC implementations allow eight coprocessors to be attached to a single microprocessor chip.

ory or to the processor. The system presently is designed to operate with two floating-point coprocessors attached to a single processor. All three operate concurrently. The design also allows for the addition of up to six additional coprocessors. Each of the two coprocessors has a separate operation code field in the instruction word. The coprocessor interrupts the processor on a floating-point exception. The microprocessor then determines the cause of the exception by reading the coprocessor status register, and the appropriate software handler is executed by the microprocessor. The coprocessor monitors the processor's status bits and uses this information to track the pipeline-stage sequencing in the processor to detect wait states and exceptions that may affect the operations within the coprocessor.

All coprocessor instructions have a fixed execution time with no operand dependencies. This simplifies the reorganizer's scheduling task. The predictability of coprocessor operations and the presence of instruction prefetching simplify pipelining. Instruction prefetching means that a new instruction word is fetched on the last cycle of the previous instruction. Two arithmetic instructions are loaded into the coprocessor with one instruction word, so that the second instruction can be initiated immediately upon completion of the first. The control signals for the operating cycles of each coprocessor are decoded one cycle ahead, eliminating control decode time and making the cycle time dependent only on the data path.

This coprocessor design is divided into two sections—the bus interface unit and the arithmetic unit—each with separate control. The chip area devoted to control of the coprocessor is very small, consisting of two small PLAs and using only 20 percent of the transistors on the coprocessor. The bus interface unit performs conditional branching and testing and monitors the instruction bus while the arithmetic unit is processing data. This architectural approach allows the CPU to execute a "branch on coprocessor busy" operation rather than executing No-op instructions. This approach also enables a branch based upon the results of the first arithmetic instruction while the second instruction is executing in the coprocessor.

The arithmetic unit is optimized for floating-point operations and provides full double-precision data paths. Eight double-precision operand registers are also available. Operands are converted to single-precision values only during load and store operations. The exponent and significant processors of the arithmetic unit are separate and operate in parallel. A single- or double-precision add or subtract operation requires four clock cycles to complete, plus two additional clock cycles if normalization is required. The multiply operation is performed 2 bits at a time, requiring 15 cycles to complete a single-precision multiplication and 30 cycles to execute a double-precision multiplication. The coprocessor hardware supports four rounding modes from the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, as well as six exceptions.

This coprocessor implementation appears to be very efficient and should demonstrate a substantial execution rate.

Silicon microprocessor designers have capitalized on the availability of large numbers of gates on VLSI chips to create complicated architectures based on parallelism and a complex set of assembly-language instructions. A gallium arsenide implementation of a microprocessor requires an alternate design approach with the availability of a limited number of gates and very fast transistors. Our study of microprocessor architectures demonstrated that a simple design, implementing in hardware only a small set of frequently used assembly-language instructions, is better suited to a GaAs fabrication technology than is a more complex design. More complicated functions are implemented in software tightly coupled to the hardware. Several computer architectures with these characteristics, known as RISCs, have been developed in the past few years. A review of these architectures indicated that one of them, the MIPS machine developed by Stanford University, could be fabricated in GaAs with approximately 10,000 gates and may operate at a clock rate of 200 MHz with a sustained throughput of 100 million instructions per second. Digital GaAs technology can take advantage of the fact that the MIPS architecture is very simple and relies heavily upon an instruction pipeline and a sophisticated assembler.

The original goal of our microprocessor evaluation project was the identification of a 32-bit microprocessor architecture that could not only execute at high speed but could also be entirely implemented on a single GaAs chip (with the exclusion of the floating-point coprocessor) in the near future. Because the MIPS architecture appeared to offer this possibility, two GaAs implementations of this architecture are presently under development,[11,12] as is the required software support. The DoD-sponsored MIPS will be able to function both as a stand-alone computer and also as a microprocessor embedded within more complex processors fabricated either in silicon or GaAs, as in the DARPA/Air Force AOSP.

Acknowledgments

This research was supported in part by contracts MDA-903-84-C-0324, N66001-85-C-0337, and F29601-84-C0016 from the Defense Advanced Research Projects Agency. The authors wish to thank C.L. Bates, D.D. Endry, S.M. Karwoski, L.M. Krueger, J.M. Ryan, M.L. Samson, D.J. Schwab, B.R. Shamblin, R.L. Thompson, C.R. Treder, T.L. Volkman, and S.K. Zahn of the Mayo Foundation for technical assistance; S. Roosild of DARPA and S. Karp for helpful discussions; and E.M. Doherty and S.J. Richardson of the Mayo Foundation for preparation of text and figures.

References

1. B.K. Gilbert et al., "Signal Processors Based Upon GaAs ICs: The Need for a Wholistic Design Approach," *Computer*, Oct. 1986, pp. 29-43.
2. *Military Standard Sixteen-Bit Computer Instruction Set Architecture*, MIL-STD-1750A (USAF), US Gov't Printing Office: 1980-603-121/3202, Department of Defense, Washington, DC, May 21, 1982.
3. *VHSIC Data Processor Unit Integrated Circuit Specification*, Preliminary Draft, Revision C, Texas Instruments, Dallas, Tex., Feb. 21, 1985.
4. M.G.H. Katevinis, *Reduced Instruction Set Computer Architectures for VLSI,* doctoral dissertation, Univ. of California, Berkeley, Calif., Report No. UCB/CSD 83/141, Oct. 1983.
5. G. Radin, "The 801 Minicomputer," *IBM Journal of Research and Development,* May 1983, p. 237.
6. I. Barron et al., "Transputer Does 5 or More Mips Even When Not Used in Parallel," *Electronics,* Nov. 17, 1983, p. 109.
7. S.A. Przbylski et al., "Organization and VLSI Implementation of MIPS," Tech. Report No. 84-259, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., April 1984.
8. J.L. Hennessy et al., 'Hardware/Software Tradeoffs for Increased Performance," Tech. Report No. 228, Computer Systems Laboratory, Stanford Univ., Feb. 1983.
9. J.L. Hennessy et al., "Design of a High Performance VLSI Processor," Tech. Report No. 236, Computer Systems Laboratory, Stanford Univ., Feb. 1983.
10. J.L. Hennessy et al., "MIPS: A VLSI Processor Architecture," Tech. Report No. 223, Computer Systems Laboratory, Stanford Univ., June 1983.
11. T.L. Rasset et al.,"A 32-Bit RISC Implemented in Enhancement-Mode JFET GaAs," *Computer*, Oct. 1986, pp. 60-68.
12. E.R. Fox et al., "Reduced Instruction Set Architecture for a GaAs Microprocessor System," *Computer*, Oct. 1986, pp. 71-80.
13. T. Gross and R. Firth, *Core Set of Assembly Language Instructions for MIPS-based Microprocessors*, Version 3.2, Software Eng. Inst., Pittsburgh, Pa., Nov. 18, 1986.

Barbara A. Naused has been a computer design engineer at the Mayo Foundation's special-purpose processor development group since 1981. Her experience includes signal processor architecture systems and GaAs integrated circuit design, comparative reviews of microprocessor architectures, and hardware and software design.

Barbara Naused received her BS degree in applied mathematics from the University of Wisconsin in 1981 and is currently completing an MS degree in electrical engineering from the University of Minnesota.

Barry K. Gilbert is staff scientist and director of the special purpose processor development group at the Mayo Foundation. He is currently responsible for the design and development of computer-aided engineering tools and hardware design methods to exploit the speed performance of GaAs digital ICs in the next generation of high-performance signal processors. His interests include the development of algorithms for the real-time analysis of wide-bandwidth image and signal data and the design of specialized computers to execute these tasks.

Barry Gilbert received his BS degree in electrical engineering from Purdue University in 1965 and his PhD in physiology/biophysics from the University of Minnesota, Minneapolis, in 1972.

Questions about this article may be directed to Barbara Naused at the Department of Physiology and Biophysics, Mayo Foundation, Rochester, MN 55905.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 150 **Medium** 151 **High** 152

FEATURE

A Distributed System for Real-Time Applications

Eli T. Fathi, Applied Silicon Inc. Canada
Eloi Bosse and Jean Caseault,
Communications Research Centre, Canada

This versatile network uses multiple microprocessors and a split-bus architecture to collect and process real-time data from a variety of sources and then transfer it to different destinations.

Modern communication applications need computer system architectures that can collect, process, and distribute large amounts of digital data. The amount of data in the system directly impacts the type of processing the system can handle as well as the system bandwidth. The handling of the digitized data (namely the collection, preprocessing, combining, postprocessing, and distribution of this data) requires complex automated procedures and special-purpose architectures.

To acquire and process data in an effective fashion, one must develop a versatile distribution network that is transparent to the system computer architecture. Such a network must not interfere with the natural flow of data in the system but rather support it throughout the various stages, from initial acquisition through postprocessing operations.

System functional requirements

The underlying motivation behind the development of the distribution system architecture we describe was our need for a general-purpose testbed for radar applications. The most important factors influencing the design of this system are the constraints imposed by the radar application itself and the current state of microprocessor technology. The radar system produces large amounts of information coming from different sources (refer to Figure 1). All this information must be directed simultaneously to multiple destinations.

The volume of data, and the type of data processing, made it impossible for any currently available (at the time the design was undertaken), single-chip microprocessor to handle the complete preprocessing load alone. We considered the possibility of using bit-sliced LSI devices, with a microprogrammed control store and lookup tables, to provide the first level of data processing. However, we decided on a more comprehensive, general-purpose solution that splits the overall processing requirement into a number of well-defined functions (for example, doppler filtering, averaging). Dedicated hardwired logic could then be used to execute these functions. In addition, it was possible to use a powerful processor to aid in the computation of lengthy functions. Additional processing power might be needed mainly to coordinate the various activities.

The system must be capable of processing large blocks of radar data in real time. It must therefore be powerful, fast, and versatile to meet the experimental requirements. It must also be capable of being easily modified or enhanced. In considering its implementation, we investigated both single processor- and multiprocessor-based system designs. We found the

0272-1732/87/1000-0021$01.00 © 1987 IEEE

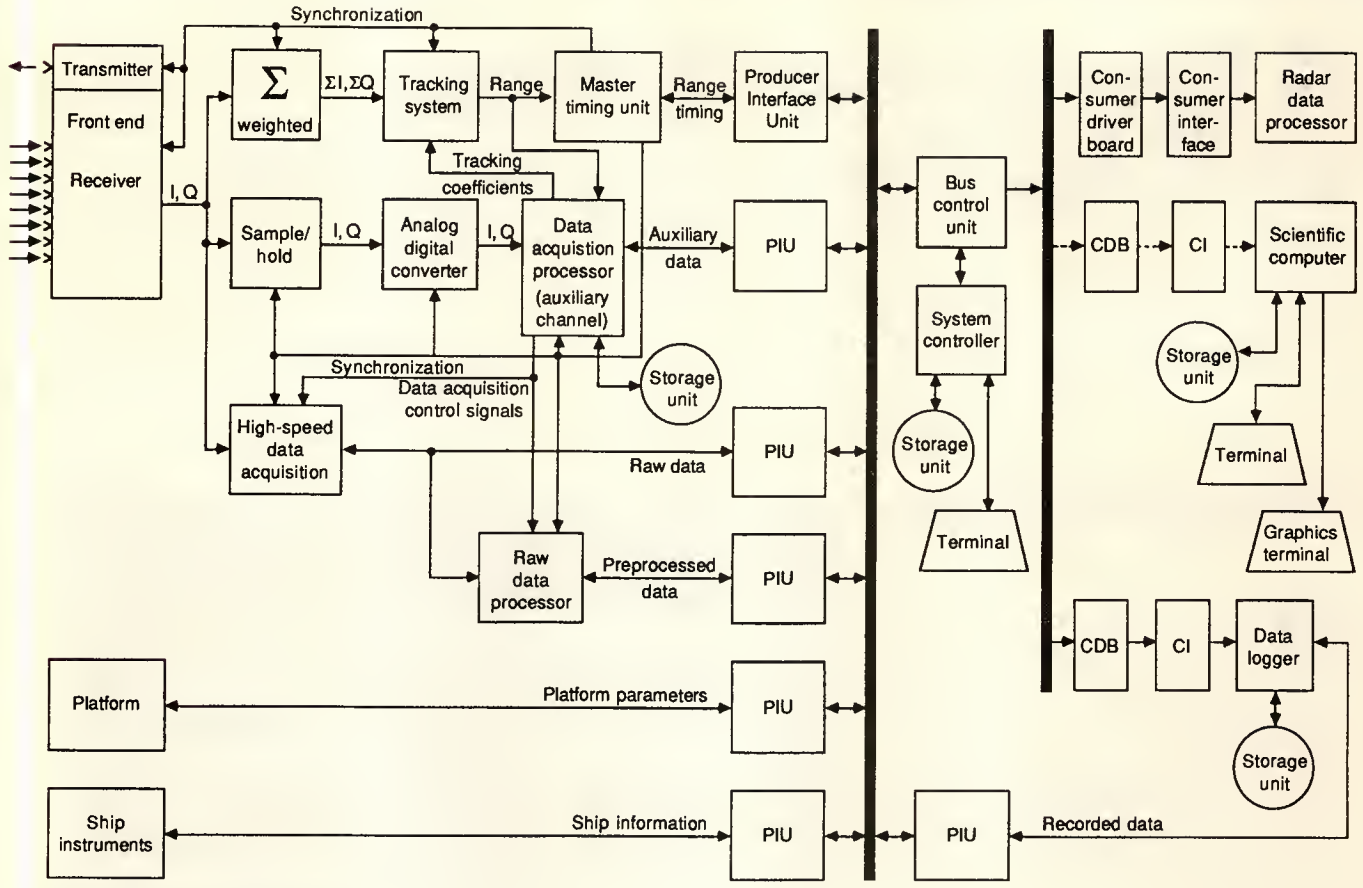


Figure 1. System architecture.

uniprocessor approach unattractive because of its poor cost-to-performance ratio, its inflexibility, its inability to accommodate future changes and/or enhancements, and the low reliability factor associated with a single device.[1] We identified the multiprocessor approach as the most suitable choice for meeting the most critical requirements of the application.

To provide the real-time processing capability and to maintain system flexibility, we chose a multilevel tree structure as the basic computer architecture. This architecture was decomposed into three major sections: the preprocessor section, the distribution network, and the postprocessor section. By decomposing the system in this fashion and providing the appropriate infrastructure, one can alter the functionality of the system by changing either the front-end or back-end portions. For example, the radar sensor could be replaced by one that operates in a different frequency band. Also, a new special postprocessor could be added in parallel with the existing processor without the need to modify hardware or software.

Some of the inherent benefits associated with this tree architecture are:

- A reduced number of processing elements across the various levels in the tree.
- Decreased software complexity, as units are self-contained, dedicated processing elements, each with its own modular software.
- A potential increase in allocated processing time, due to possible data reduction as data moves through the various levels.
- A reduced number of soft errors accumulated since processing is done near the point of acquisition.
- Fail-soft capability, as real-time diagnostics routines can be performed between adjacent levels. The system is capable of masking out bad data as a result of software errors and/or hardware malfunction.
- Accommodation of new devices such as faster microprocessors and special-purpose LSI/VLSI hardware modules at a later date.

Distribution network architecture

We based the design of the system distribution network upon Anderson and Jensen's study of interconnection transfer strategy, control method, and path structure.[2] In particular, we placed emphasis on studying the system architecture by considering the type of communication allowed between the various modules rather than the structure of the modules themselves.[3]

We developed an interconnection structure that approached the design of the transfer strategy by combining the best features of the direct method (such as is found in single-bus architectures) with the best features of the indirect method (such as is found in star interconnection structures). This hybrid interconnection structure employs a single bus for the input devices and another single bus for the output devices. The use of two separate buses is dictated by reliability considerations,[4] as well as speed of operation. Between the two buses, a bus controller effectively provides a star configuration[5] by supporting centralized routing (transfer control method) with dedicated paths (transfer path structure) to the two buses. We term the system architecture, which is the result of this unique combination of different transfer strategies, a double star configuration.[6]

The whole distribution network can be considered as a loosely coupled multiprocessor system with each processor having its own local program and data memory as well as I/O. In addition, some of the processors share a common communication memory to facilitate the fast transfer of data. By using multiple processors, we reduce the system software complexity significantly. Each processor has its own simple executive to manage the resources and transactions within its domain. Thus, by increasing the amount of hardware, we eliminate the need for a complex real-time, multitasking executive, which would have been required on a uniprocessor system.

The hub of the system is a dual-processor subsystem consisting of a master minicomputer and a slave microprocessor. This dual-processor configuration initiates, monitors, and controls all the transactions occurring within the domain of the distribution network. We refer to the minicomputer master as the system controller, or SC; the specially designed microprocessor slave system is the bus control unit, or BCU. Aside from the SC, the distribution network supports two other types of devices: *producers*, which output data to the distribution network and *consumers*, which remove data from the distribution network.

This interconnection scheme contains two main buses that are physically isolated from each other via a special bus control unit (see Figure 1 again). This physical separation matches well the logical separation between producers and consumers. By following the dual-bus approach, the consumer and producer parts of the system can be expanded independently and selectively. Thus, if additional producing devices must be added to the system, the physical characteristics of the consumer devices bus are not affected and vice versa. Furthermore, the addition of more consuming devices on the bus would require minimal software/hardware changes to the BCU.

Each of the producing devices connects to a producer interface unit, or PIU, to match its own characteristics (serial, parallel, analog) to those of the producer bus (see Figure 2). This permits simultane-

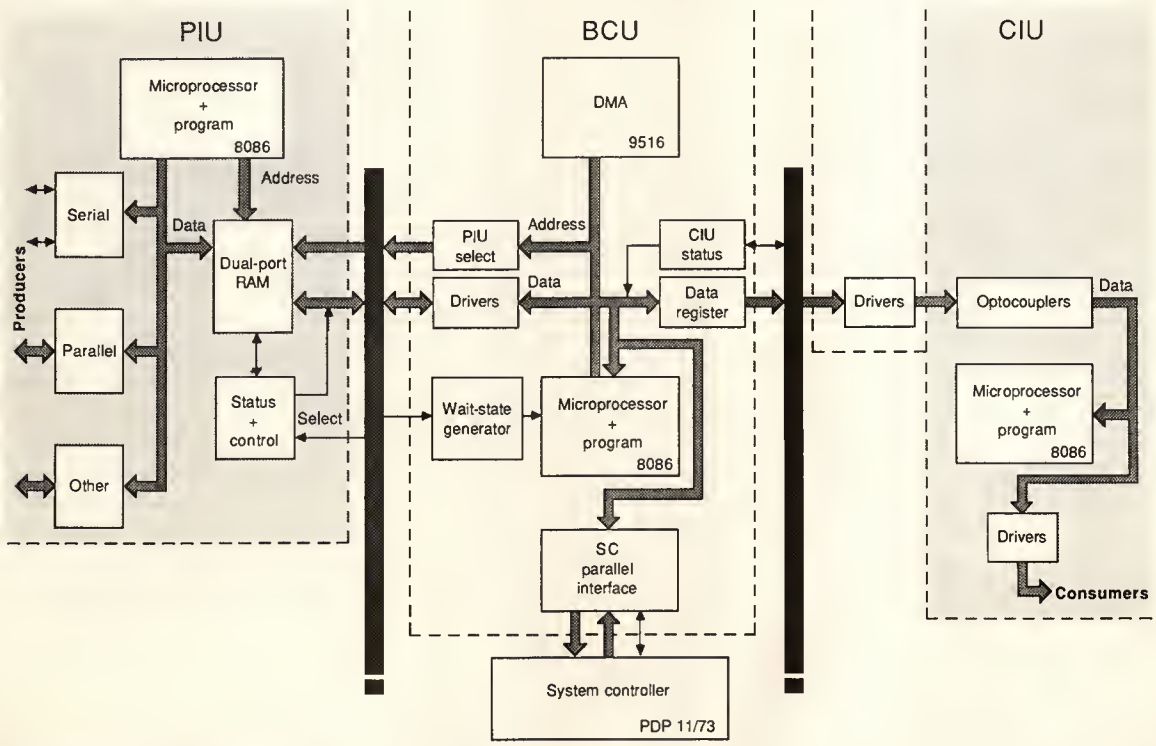


Figure 2. Distribution network.

ous reception of data from each producing device. The transfer of data from the producers to the consumers, and possibly between producers, is initiated and controlled via the BCU. To facilitate this transfer, we designed a bidirectional producer bus.

The consumer bus is a unidirectional, high-drive bus with multiple receivers that are electrically isolated from the producing devices. Any interested consumer has the capability of picking up any or all of the data associated with a given block via its consumer interface unit, or CIU.

System controller

In the proposed configuration, the system controller is the hub of the network; its main control functions are monitoring the system, initiating control functions, and interfacing with the operator. The BCU, acting as a slave to the SC master, controls and monitors the distribution network. The SC communicates with the BCU via a dedicated parallel interface using a single encoded command word. With this command word, the system controller can send or receive information or commands to/from any producer, consumer, or the BCU. The BCU accepts a command, decodes it, executes it, and sends the obtained information back to the SC.

We maintained a very simple communication protocol between the SC and the BCU. While the BCU executes a command, it can be interrupted by, at most, one more urgent command at a time. In addition to the basic command word, a special command word is used by the BCU to notify the SC of an urgent request.

Since all commands must originate either from the SC or be approved by it, the SC, at any one time, has the exact global picture of the whole system. The system operator, through the use of the system monitor, can easily determine the status of the system and control its operation by initiating the appropriate command on the keyboard.

Bus control unit

The SC, being an off-the-shelf minicomputer, lacks the appropriate hardware modularity and software flexibility to manage the complete distribution network by itself. Aside from the requirement for multiple interfaces to multiple sources and destinations, the system sets a stringent requirement on the data transfer rates (a high data transfer rate per device, coupled with data gathering from multiple devices). These conditions made the performance unattainable for a single off-the-shelf device. Therefore, we had to develop a tailor-made subsystem that could accommodate the various devices involved in the transaction, by interfacing to the SC on one side and to the remaining devices on the other. This led us to develop the BCU. We designed the BCU as a slave to the SC, having it execute commands as well as manage the data packet transfers from the producers to the consumers.

As shown in Figure 2, the BCU subsystem consists of three main functional blocks: processing, DMA, and I/O. The processing section handles all the commands coming from the SC. The processor is responsible for functions such as error detection, special-purpose data transfers, control and status transfers, and communication with the system controller. The BCU's main task is to perform high-speed data block transfers. To accomplish this, it needs a DMA to meet the speed requirements. When the BCU is interrupted, the DMA gains control of the bus, picks up information from various producers, and latches it onto the consumer bus at maximum speed.

Producer interface unit

The producer interface interconnects the producing devices and the distribution network. The data generated by individual producers is collected by a producer interface unit, or PIU, which is capable of simultaneously servicing all producers connected to it. Each PIU must provide the appropriate interface to match the characteristics of each producing device, assemble data in a special format for the BCU, accept information from the SC via the BCU, and send it to its producing devices.

Each PIU can service many producers having different input/output characteristics. In the current design, the PIU supports the following interfaces: one multipurpose parallel interface, two RS-232C type serial interfaces, and one special-purpose interface that is accommodated via a reserved wire-wrap section. The multipurpose parallel interface has a flexible structure; a number of programmable control lines to synchronize data transfers are provided. Also, data transfers can be supported either by an onboard FIFO or directly by the PIU's processor.

The main component of the PIU is a true dual-port read/write memory. This memory can be accessed simultaneously by both the BCU and the PIU and is used for transferring control and data information between them. Since multiple PIUs could be connected to the common bus, bidirectional tri-state buffers are provided on the outputs of the dual-port memory. In the event that both the PIU and the BCU attempt to access the same control memory location simultaneously, the BCU receives priority, and the PIU access is delayed until the BCU has completed its operation.

Consumer interface unit

The BCU transfers information packets collected by the PIU at high speed to the consumers. The in-

formation goes to every consumer. Each consumer has a consumer driver board connected directly onto the consumer port (Figures 1 and 2). Each driver board sends the information via twisted-pair cables (differential voltages) to the consumer receiver board (which isolates the signals using optocouplers). Data is transferred to the consumer interface unit (which provides the intelligence to sort the received information). The large distances involved and the large number of consumer devices (up to 10) forced us to minimize the number of control lines and to select a communication protocol that was independent of the consumers. The communication protocol between the BCU and the consumers permits continual availability of the data on the bus, together with its clock. The BCU asserts the block transfer start signal and a clock signal at the middle of each data word transfer.

Any interested consumer may access this data; however, a consumer cannot interrupt the transfer. So to verify consumer status, we provided three status lines, as well as one request line, for the consumer to indicate an exception condition. Whereas each consumer has a dedicated request line, we kept the three status lines common to all consumers. If one or more request lines are asserted, the BCU selects the appropriate consumer request condition by enabling its status buffer.

Data block transfers

To transfer large blocks of data from different producers, we built a distribution network with a variable memory organization and shared banks of data between the BCU and the PIUs. Many special features were incorporated into the design to facilitate the address allocation to each data word: bank switching, shared addressing, programmable selection of the PIUs, and the DMA "latch while reading" option.

In general, each PIU collects data from its producers. When a PIU is interrupted by a particular producer, the producer's data is immediately stored at its allocated address, inside the data block *x*.a (*x* is the PIU number) of the dual-port RAM (shared by the BCU and the PIU, see Figure 3). Once the entire group of data that this particular PIU has to collect fills the allocated addresses inside the data block, the PIU switches the data block flag bit. Next the PIU begins writing into data block *x*.b, allowing the BCU to read block *x*.a. By reading (BCU) and writing (PIU) different blocks, the system has no need for a complex control logic. The PIU works at maximum speed, not interfering with the BCU. The BCU reads in the latest available complete data block, always knowing that the data is valid. The PIU sets up a data block flag bit to indicate when the data has been updated.

After the data has been sorted by the PIUs, the BCU can initiate data block transfers. This is done on a regular basis (2 KHz). When the BCU receives a start signal, the DMA takes control of the bus, by supplying addresses and latching data words directly into the consumer port during the read cycle. Thus the DMA allows the bus to work at twice the speed (the current speed is 16M bits/s, but we expect it to double in the final system).

The data sent to consumers contains two types of information: control words like the block type, status, and information coming from the BCU memory, and the data block itself, collected from various PIUs. The data block size is completely programmable; it contains a variable number of control words (BCU memory) plus a variable number of data words (PIU dual-port RAM). Because the BCU memory is adjacent to the PIU data, the DMA easily transfers the BCU data, and afterwards, the PIU data. We added a special feature to the BCU to increase the flexibility of the block transfers: The PIU selects data on a word basis, not on a block basis. Therefore, a data block is a selection of independent words, each one obtained from any PIU. The block, as a whole, becomes a group of PIU data packets (seen in Figure 3). To make this possible, every BCU data block configuration must provide its own sequence of PIU selection. Up to 14 block configurations can be programmed, with immediate access to two of them by the DMA (odd blocks on the first channel, even ones on the second) and to the others by changing the starting address inside the DMA.

Finally, when the DMA has put the data on the bus, and has latched it onto the consumer port, the consumer driver board amplifies the signals and drives them to the consumer interface unit, 200 feet away, at 16M bits/s (projected speed of 32M bits/s). The CIU, after receiving the block, decodes the block type word and the status word (status of PIU data) and decides whether it should keep the block or not. If the test is positive, the data can be distributed to its allocated consumer, at the consumer's speed.

Figure 3 shows the configuration of the dual-port memories of each PIU. The design incorporates two levels of memory selection. The PIU controls the first level and forces the BCU to read a predetermined PIU data block (*x*.a or *x*.b, *x* being the PIU number) while the PIU is writing into the other (*x*.b or *x*.a). The BCU sees only one PIU data block, but the PIU selects the one (a or b) to be read by the BCU. The second level of memory selection is located on the BCU side. The PIU-selected logic specific to each BCU data block contains the PIU number associated with each data word inside the BCU data block. The resulting BCU data block constitutes an arrangement of words coming from different PIUs according to the predetermined pattern of selection. This method permits up to 14 BCU data blocks, seven of which are composed of PIU data block type one and seven, of block type two.

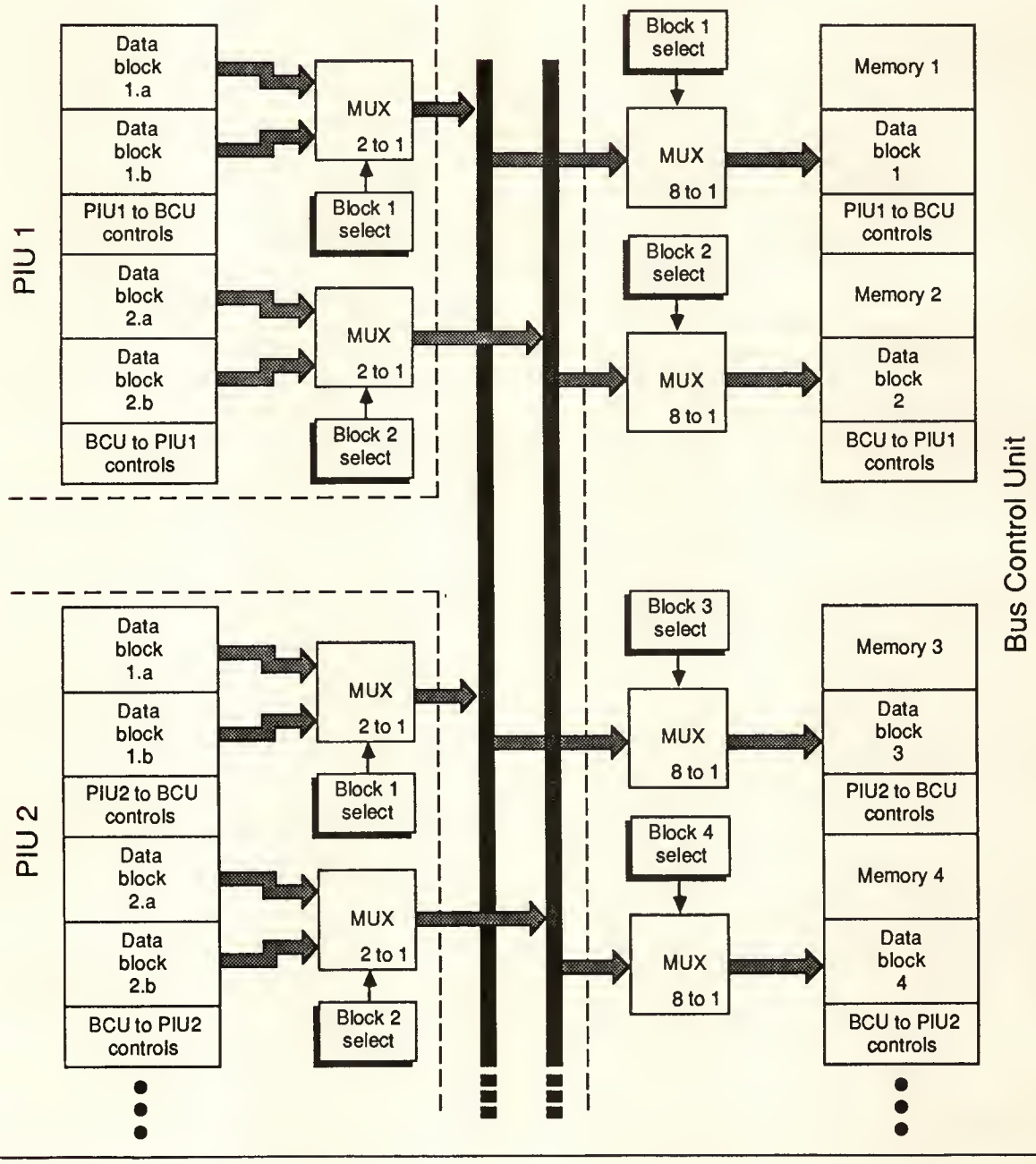


Figure 3. PIU/BCU shared memory.

BCU software: command execution

When the BCU is not interrupted by the data block transfers, it executes the commands requested by the system controller (see Figure 4). One register controls every transfer during the execution of a command: the *control* register defines when the operation level is active, when the system controller interface is active, and when the BCU transmits to the destination all of the information received from the source. One feature of the command execution software is that it can keep track of special requests while the system controller interface is busy. It permits the BCU to detect system errors and interrupt the command execution.

A transaction involving the system controller (which might be the source or the destination) requires two steps. First it obtains information from

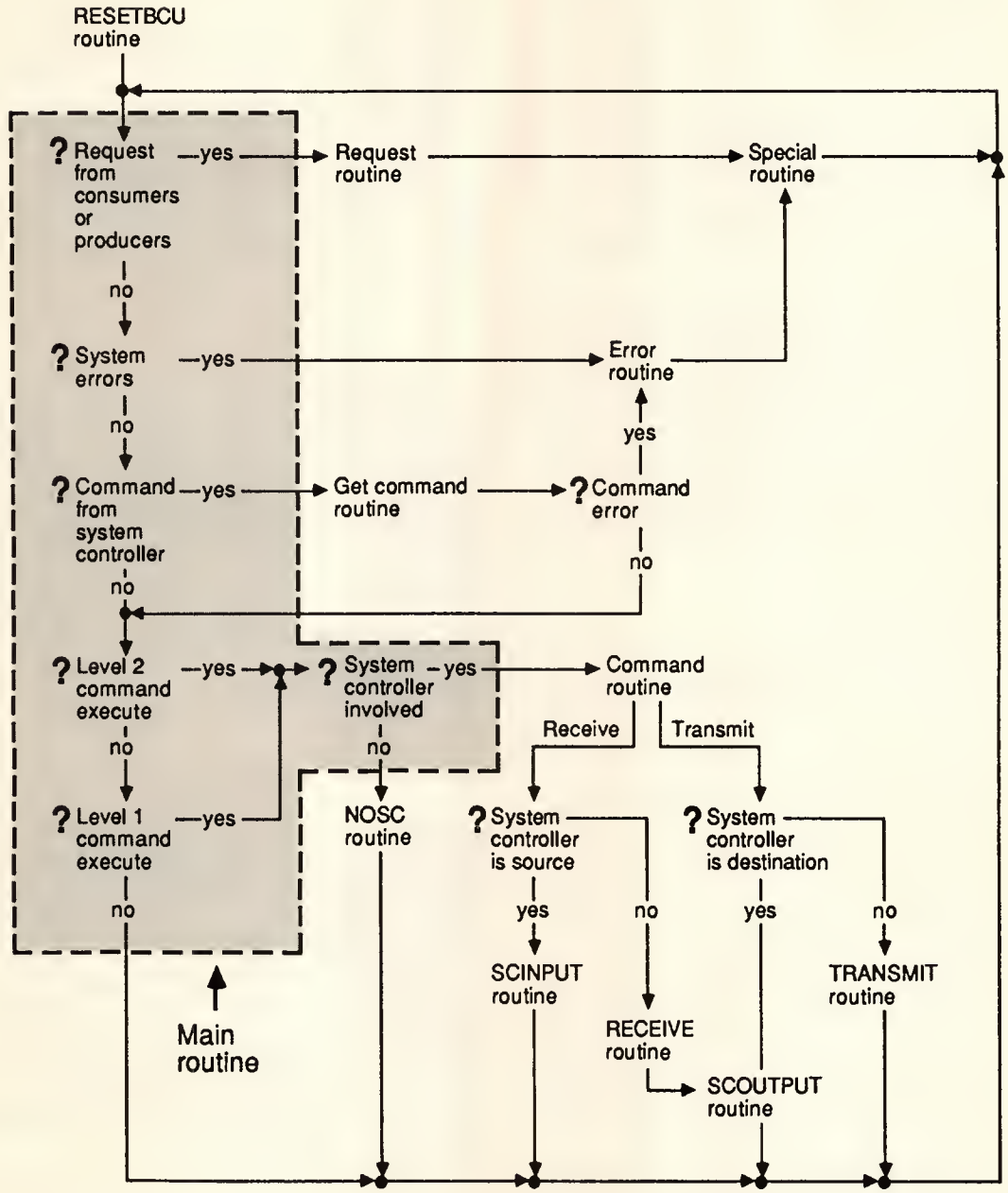


Figure 4. BCU software flowchart.

the source and stores it in the BCU memory. Next it sends this information to the destination. After receiving the command word, the control register activates the current operation level. The command word is decoded by the BCU. If the SC is not involved in the operation, the NOSC routine completes the execution of the command. Otherwise, the COMMAND routine initiates the first half of the command: the reception of the control words from the source to the BCU memory. The SCINPUT routine (SC is the source) or the RECEIVE routine (SC is the destination) receives the control words. When reception is finished, the COMMAND routine is called again by the MAIN routine to finish the command execution with either the SCOUTPUT or the TRANSMIT routine.

We have described a distribution network that facilitates reliable, high-speed data transfers for systems with multiple sources and destinations. To support this goal, we developed an elegant system bus structure, which we call a double bus star interconnection. Such a structure provides many benefits derived from both the single bus as well as the star type of routing mechanism. It shows great promise for expansion to multibus star structures for other applications that could be characterized by the producer/consumer model described here.

We intend to continue improving the system interconnection structure as technology develops. The system bandwidth is directly related to these improvements, as is the speed of operation of the bus control unit. The BCU, at present, handles all the DMA control, but in later improvements the other modules on the buses could conceivably assume the DMA function. In addition, we also plan to consider increased data rates, data quantity, and data transmission distances.

Acknowledgments

We thank Gordon Marwood, Ross Fines, and Denis Lamothe for their contributions to this project.

References

1. E.T. Fathi and M. Krieger, "Multiple Microprocessor Systems: What, Why and When," *Computer*, Mar. 1983, pp. 23-32.
2. G.A. Anderson and E.D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics, and Examples," *Computing Surveys,* Dec. 1975, pp. 197-214.
3. K.J. Thurber et al., "A Systematic Approach to the Design of Digital Bussing Structures," *AFIPS Conf. Proc.,* 1972, Montvale, N. J., pp. 719-740.
4. D.R. Powell, "Dependability Evaluation of Communication Support System for Local Area Distributed Computing," *Proc. 12th Ann. Int'l Symp. Fault Tolerant Computers,* IEEE, New York, June 1982.
5. C. Weitzman, "Distributed Micro/Minicomputer Systems, Structure, Implementation and Applications," Prentice-Hall, Englewood Cliffs, N.J., 1980.
6. E.T. Fathi and N.R. Fines, "Real-Time Data Acquisition, Processing, and Distribution for Radar Applications," Real-Time Symp., Computer Society of the IEEE, Washington, D.C., Dec. 1984.

Eli T. Fathi is president of Applied Silicon Inc. Canada. He is currently working on the development of application-specific integrated circuits for diagnostics and testability. His main research interests are in the areas of advanced computer architectures and high-speed DSP modules.

Fathi received his BASc and MASc degrees in electrical engineering from the University of Ottawa in 1978 and 1981. He is a member of the Association of Professional Engineers of Ontario and of the IEEE.

Eloi Bosse is a research engineer for the Communications Research Centre, where he is working on developing leading-edge algorithms for radar signal processing. High-performance distributed computer architectures for real-time applications are his main research interests.

Bosse received a BScA and MSc in electrical engineering from the Universite Laval de Quebec in 1980 and 1981. He is a member of the Ordre des Ingenieurs du Quebec.

Jean Caseault is a senior design engineer for Applied Silicon Inc. Canada. When he worked on the project reported here, he was a development engineer for the Communications Research Centre in Ottawa, Canada. He has worked on a number of sophisticated microprocessor-based systems for an experimental radar. His interests are in the areas of distributed computer architectures, special-purpose controllers, and VLSI chip development for DSP applications.

Caseault received a BScA in electrical engineering from the Universite Laval de Quebec in 1982 and is a member of the Ordre des Ingenieurs du Quebec.

Questions regarding this article can be directed to Eli T. Fathi, Applied Silicon Inc. Canada, 310-2255 Ft. Laurent Blvd., Ottawa K1G 4K3, Canada.

Reader Interest Survey

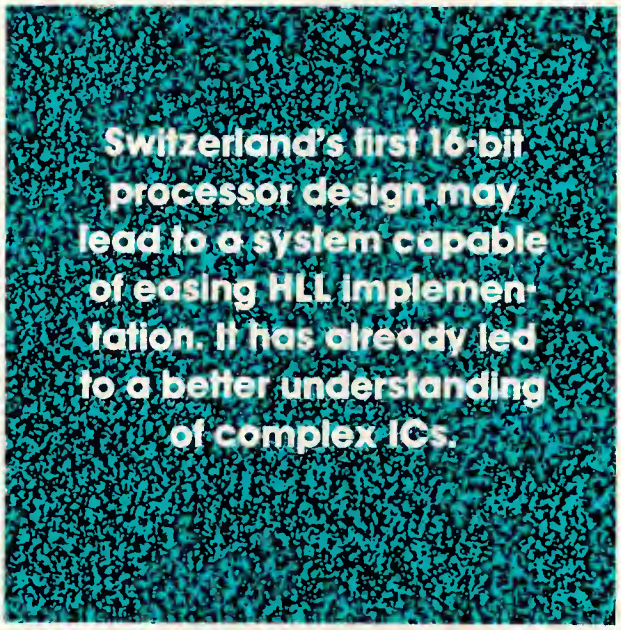
Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 162 Medium 163 High 164

FEATURE

A General Heap Processor

Eduardo Sanchez, Patrick Sommer, Jacques Menu, and Christian Iseli
Ecole polytechnique federale de Lausanne



At the Ecole polytechnique federale de Lausanne in Switzerland we have recently completed the design of a 16-bit processor in two versions, bit-slice and VLSI. The goals determined at the beginning of the project were to:

- acquire a new capability (no logic design project of such a large scale had ever been undertaken in the school);
- develop design tools (microassemblers, simulators, graphic editors) that would be usable for teaching and research in the future;
- encourage software and hardware teams to cooperate in finding principles common to languages, to architectures, and to design methodologies;
- test, on a complex circuit, the integrated circuit design methods that the school developed in cooperation with the Centre suisse d'electronique et de microtechnique de Neuchatel (CSEM, Switzerland);[1] and
- obtain as a final product a processor that would ease implementation of high-level languages and particularly of the Newton language developed at the school.[2]

We chose to facilitate the implementation of the following high-level-language characteristics:

- dynamic data structures management;
- coroutines and procedures management, namely the allocation of their activation blocks;
- detection and handling of runtime errors (overflow, use of uninitialized variables, nil pointer dereferencing); and
- debugging facilities.

These choices led to a quite classical, microprogrammed processor architecture. However, we organized data memory management in the form of a *heap,* which is completely ruled by dedicated instruction.

Memory management

Most of the actual high-level languages are block-structured languages.[3] A program is decomposed into blocks, called procedures or routines (within which local variables can be declared) known only in their environment.

This property, present in Pascal for example, has consequences for the physical implementation of the language. A memory allocation for a local variable is necessary only when its procedure (the block which contains it) is active. However, this economy of memory leads to a problem: The zone of the memory that contains the local variables cannot be statically allocated. The classical solution to the problem is the implementation of the data memory in the form of a stack.[4] In that way, to each call of a procedure, designers allocate a block of memory placed at the top of the stack (the activation block) for its local variables (the stack increases). When the procedure ends its execution, the memory block becomes free (the stack decreases) and can be used again by another procedure. The memory's location is then used in the optimal way.

When one is confronted with dynamic variables (managed by the New and Dispose procedures of Pascal), management of the memory is a little more complex because the variables can live longer than (or can survive to) the procedure that has created them. The management of the coroutines leads to the same problems.

0272-1732/87/1000-0029$01.00 © 1987 IEEE

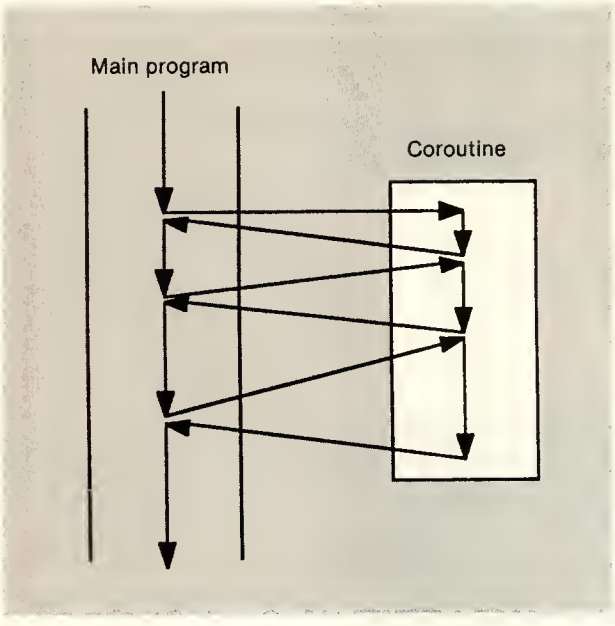


Figure 1. Execution of a coroutine.

The essential difference between a procedure and a coroutine is the following. When a procedure is called, it is executed from beginning to end; that is to say, with only one entry point and one output point. However, a coroutine can be stopped at any point and control goes back to the routine that has called it. At the next call, the coroutine comes back to the point from which it left, keeping its previous state (Figure 1). The stack is no longer a satisfactory solution for the problem of memory allocation for variables, since the memory blocks are no longer necessarily released in the inverted order of their allocation.

As a result we resorted to a heap form of managing memory. Blocks of the memory are allocated at each call of the procedures or of the coroutines and at each creation of dynamic variables. The blocks of the heap are sequentially allocated from one extremity of the memory, and the heap increases in the direction of the other extremity of the memory. As soon as one has finished using a block of the heap, one marks it as being free. The heap then composes a suite of free and occupied blocks, in an aleatory order. As the blocks are sequentially allocated, a moment arrives when the heap occupies the whole memory. One then needs a garbage collector that compacts the blocks occupied on one side of the memory and the free blocks on the other. The allocation of new blocks begins again at the last occupied block.

The notion of general heap leads to only one philosophy of memory management, although one generally finds a stack and a heap in the usual implementations of high-level languages.

Organization. The entire data memory available to the processor, lying between the *heaplow* and *heaplim* addresses, is organized in the form of a general heap: memory blocks are allocated through a basic instruction named Allocate.

As seen in Figure 2, the blocks are linked together, forming two types of chains:

- *The process chain*. The first block is the system block and the last is the current process block. Between these two blocks are all the blocks of the processes that have been suspended but can be reactivated by the program.
- *The procedure chain*. Each process has its own procedure chain, which is organized similarly to the stack of the usual high-level-language implementation.[4]

It is guaranteed that any data created at some place in a program will remain alive as long as it is needed by the execution of that program. In particular, there is no means to destroy data explicitly (compare Dispose in Pascal), and dangling references to destroyed data cannot exist. When a block is no longer useful, the space it occupies is automatically recovered at the next garbage collection/compaction.

Any reference at a memory element is made through a double pointer named *heep*. The first word of a heep is named the address part and contains the initial address of a block in the heap. The second word, which is named the complement part, contains additional information, generally as an offset inside the block.

The state of the program is held at every moment in four heep registers:

- *System*. The address part contains the address of the system block; the complement contains nil.
- *Global*. The address part contains the address of the program block; the complement contains the process type.
- *Current*. The address part contains the address of the active process block; the complement contains the process number.
- *Local*. The address part contains the address of the active procedure block; the complement contains nil.

The structure of a block. The base element of the heap is the data block. A block is requested through the Allocate instruction when a quasiparallel process (coroutine) is created, when a procedure is called, and at each dynamic data allocation (character strings, unbounded sets, rows, queues, associative tables, and so on).

Figure 3 shows the structure of blocks in memory. Each block is composed of:

- a four-word header;
- a part containing the heeps, that is, references to other blocks; and

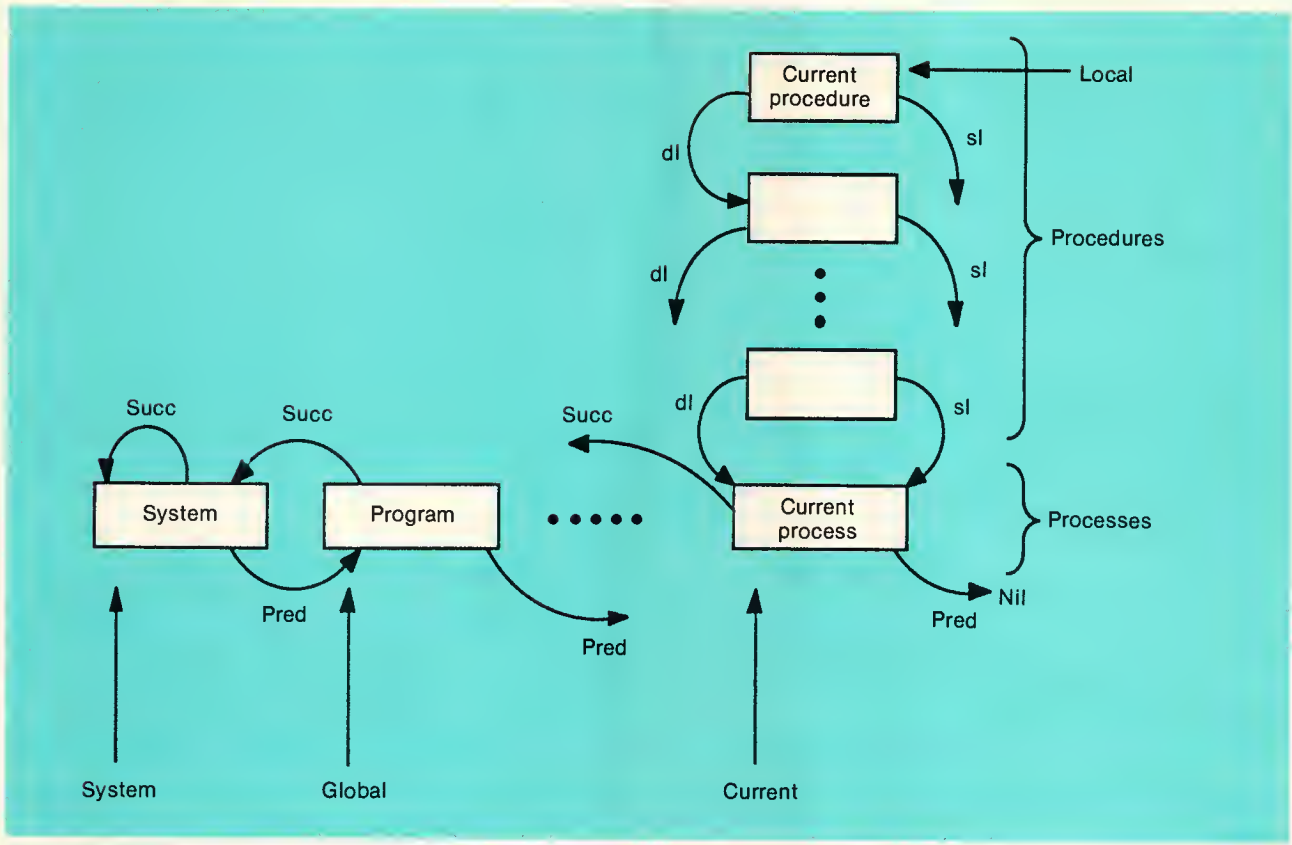


Figure 2. Block linkage.

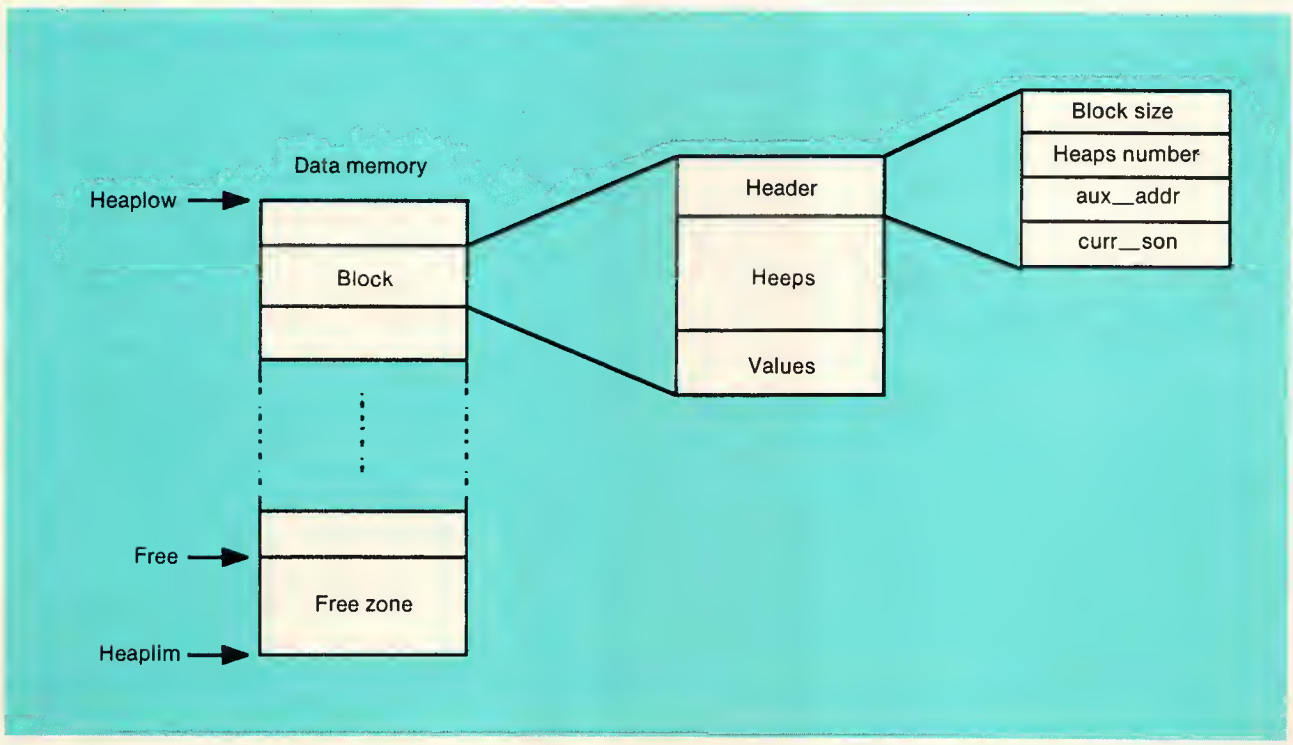


Figure 3. Memory organization.

• a self-contained data part, which groups data not referring to any other block in the heap, such as characters, Booleans, numbers, and the like.

The header does not contain program data, only information necessary to memory management. The first word gives the size of the block (in number of words), and the second word, the number of heeps. The two last words are temporary variables used during the garbage collection.

The block of a process has at least five heeps used to link this process to the process chain in the heap:

• *Static link*. The address part points to the block of the process or procedure where this process was declared; the programmer is free to use the complement part.
• *Dynamic link*. The address part points to the block of the procedure that was active at the time of the suspension of the process or calling routine; the complement part contains the return address in the code.
• *Successor link*. The address part points to the block of the process that will be executed after the current process is suspended; the complement part contains the process number.
• *Predecessor link*. The address part points to the block of the process that was executing before the current process; the complement part contains the process number.
• *Global link*. The address part points to the block of the program that owns the process; the complement part contains the process status.

Only two heeps are necessary to link the procedure's blocks, the static link, and the dynamic link.

The system block contains, in its self-contained data part, eight useful values for the debugging of programs. They are:

• free_sys_save, the address of the top of the heap (free) at the time the system launched the program. At every instant free gives the address at which the next block can be allocated on the heap;
• free_prog_save, the value of free at the time the program stopped;
• pc_save, the value of the program counter at the time the program stopped;
• ms_save, the processor status at the time the program stopped;
• current_save, the value of current at the time the program stopped;
• global_save, the value of global at the time the program stopped;
• local_save, the value of local at the time the program stopped; and
• heaplim_save, the value of heaplim, that is, the address of the top of the data memory. The size of the data memory is heaplim – heaplow.

Garbage collection. The heap may be considered as a directed graph whose nodes are blocks and whose arcs are heeps. The root block is always reachable via Local, one of the processor's heep registers. A block is said to be reachable if the graph contains a path from Local to the block. Note that circular passes can occur in the heap and that the memory manager must cope with them.

The only way for a program to indicate that a block is no longer needed is to cut an arc leading to it, for example, by assigning a distinct value (nil) to a heep that pointed to it. However such a block becomes unreachable only when the given arc is the only one leading to it. By cutting such an arc, it is also possible to make a whole subgraph unreachable, if the only references to its nodes originated from the arc just cut.

The physical data memory is divided into two logical parts (Figure 3); at one end is the heap zone, between heaplow and free, containing contiguously all the blocks currently allocated, whether reachable or not. The other end, between free and heaplim, is a free zone from which space can be fetched to allocate a block whenever requested. The heap zone grows as long as block requests can be satisfied by the free zone, moving the logical barrier toward the free end of memory.

When the free zone becomes too small to satisfy a request, the process of garbage collection begins. It successively executes the following phases:

1) Determine which blocks in the heap zone are reachable, starting from Local.
2) Compute the future address of every reachable block, that is, the address that will be assigned to it after compaction.
3) Update the heeps, replacing the current address field by the new address computed in the previous phase.
4) Move all reachable blocks to the heap end of memory, thus creating a single, new free zone.

If the block request that caused garbage collection to occur can be satisfied, the execution of the program proceeds normally. Otherwise the heap overflow is flagged and the program aborts.

The first phase (marking reachable blocks) is the only one that needs to go through the heap, starting from Local and following the arcs; the last three are merely linear scans through the heap zone. This process of going through the heap is conceptually recursive, since each heep in a block just marked points to a block that has to be processed too.

How can we handle the (implicit or explicit) stack needed to implement this recursive algorithm when the physical memory is precisely full? We do this by reserving enough space in each block to contain the origin of the arc that led to it. This reversed arc is stored in the third word of the header, named

We built the Hip1 prototype of our processor, using bit-slice circuits, around the Am29203.

aux_addr. The word is set the first time the marking phase visits a reachable node. Since this word is normally nil, a separate Boolean field to mark reachable blocks is not necessary. The last three phases recognize the blocks as having a nonnil value in aux_addr.

How in turn do we go through the subgraphs originating from a node? This we do with a For loop whose index is stored in the fourth word of the header, named curr_son.

Multiple arcs leading to the same block and circular paths in the graph are easily handled by not processing heeps contained in an already marked block.

This heap management can be seen as a generalization of the Lisp 2 algorithm described by Knuth,[5] whose performance Cohen and Nicolau have assessed.[6]

Instructions for memory management. The base instruction in memory management is Allocate *m*,*n*, which allocates a block of *m* words and which contains *n* heeps. This instruction causes garbage collection if necessary.

We provide an instruction Garbage as a convenience, so programmers can launch a garbage collection at any time.

Process management instructions. The first instruction to be executed by the processor (INIT_RUN) initializes the blocks of the operating system. To launch a user program, the system executes the instruction ENTER_PROG adr, where adr is the starting address of the program code. This instruction also begins garbage collection, to furnish the maximum free data memory to the program.

The program code, possibly generated by a compiler, must first allocate the block for the program with an Allocate instruction and then use the instruction ENTER_SECT adr to start the execution of the program at the address adr. The latter instruction also updates the processor registers and the links between the program and the system.

At the end of the program the instruction END_PROG returns control to the system.

The program may be divided into processes (or coroutines) and into procedures. The instruction ENTER_PRSS adr starts the execution of a process at the address adr and updates the processor registers and the links of the process. The process block must have been created before that by the Allocate instruction. The process execution ends with the instruction Terminate, which then activates the successor of the process.

We reactivate a stopped process with the following four instructions:

- Activate pr. The process containing this instruction will become the successor of the activated process. The pr parameter contains the address of the activated process block and the address of the code to be executed.
- Resume pr. The process containing this instruction is detached. The pr parameter contains the address of the block of the process that will be activated along with the address of the code to be executed.
- Return. Control passes from the current process to its successor. The current process is detached.
- Exchange. The current process is exchanged with its successor.

Procedure management instructions. Two instructions can be used to activate a procedure: ENTER_PROC and ENTER_NAME. The former takes the starting address of the code of the procedure as a parameter, and the latter takes the address of the block of the procedure and the starting address of the code as a heep parameter. In both cases the block of the procedure must formerly have been allocated with an Allocate instruction.

The instruction EXIT_PROC terminates the execution of a procedure.

Bit-slice implementation

We built a prototype of the processor, using bit-slice circuits, around the ALU Am29203; its name is Hip1.[7] The Harvard-like processor has two zones of memory that are physically separated:

- the program memory, which has a size of 64K words of 16 bits; and
- the data memory (the heap), which has a size of 64K − 1 words of 16 bits. The entire 64K words are not used because the address 0 is used as the value for nil (that is to say, that heaplow = 1 and heaplim = $2^{16} - 1$).

The two memories are word addressable.

Input and output are treated separately: in this case, only the low-order byte of the address and data buses is used, allowing 256 input units and 256 output units of 8 bits each.

The microprogrammed control unit uses the Am2910 sequencer. The microcode is very horizontal, allowing optimal use of parallelism; its size is 1K × 72 bits, of which 116 microinstructions are used to implement the garbage-collection algorithm.

System management
  The INIT_RUN instruction allocates and initializes the block for the operating system. It must be the first instruction to be executed when the processor is powered up.
Program management
Process management
Procedure management
Memory management
Register transfer
Loading a register from the data memory
Loading a register from the program memory
Storing a register in the data memory
Storing a register in the program memory
Register intialization
Integer arithmetic operations
Cardinal operations
Byte operations
Bit operations
Shift and rotate operations
Logical operations
Jump operations
Input/output operations
Miscellaneous operations
  In addition to the NOP operation, there are two debugging instructions:
  DUMP adr updates the system block, namely the eight values described earlier, and passes control to the debugging procedure, which starts the code at the address adr.
  ERROR #n updates the system block, stops the execution of the program, and returns control to the operating system. The value *n* is assigned to the program termination status. *n* is a user-defined error code ($16 \leq n \leq 225$; the system reserves the codes 0...15).

Figure 4. Classes of instruction.

The processor is physically realized on a double VME-format board; another board implements the interface between the processor and a VMEbus, thus allowing communication between the processor and other commercial boards like memory and interface boards.

Data types. Hip1 handles four data types: heeps, integers, bytes, and cardinals (unsigned integers). Three types of registers handle these different data:

- eight heep registers, H0 through H7. Three of these, H5, H6, and H7, are used for Global, Current, and Local addresses. A System register does not exist; the systems block always starts at address 1 (heaplow);
- 16 word registers, W0 through W15; and
- 16 byte registers, B0 through B15.

Addressing capabilities. The data may be addressed in three different ways with Hip1:

- *Indirect addressing*:

  address = Hi.a + offset

where Hi.a is the address part of the heep Hi, and offset is a 16-bit value given by the instruction. An example is

```
LOAD_INDIR      LOCAL,#12,W0
M[LOCAL.a + 12] → W0
STORE_INDIR     H4,LOCAL,#8
H4 → M[LOCAL.a  + 8]
```

- *Referenced addressing*:

  address = Hi.a + Hi.c

where Hi.c is the complement part of register Hi. Example:

```
LOAD_REFER      H4,W0
M[H4.a + H4.c] → W0
STORE_REFER     B2,H3
B2 → M[H3.a + H3.c]
```

- *Indexed addressing*:

  address = Hi.a + Wj − Hi.c + 4

where Wj is a word register and 4 is the size of a block header. This addressing mode is used to access array elements. Hi.c contains the lower bound of the indices, while Wj contains the index of the particular element to be accessed. Moreover, the register Wj may be post-incremented or pre-decremented. For example:

```
LOAD_INDEX      H2,W3,W0
M[H2.a + W3 − H2.c + 4] → W0
LOAD_INDEX      H1, (W3+),B2
M[H1.a + W3 − H1.c + 4] → B2
W3 + 1 → W3
STORE_INDEX     W1,H3,( −W2)
W2 − 1 → W2
W1 → M[H3.a + W2  − H3.c + 4]
STORE_INDEX     H0,H2,W4
H0 → M[H2.a + 2(W4 + H2.c) + 4]
```

Of course, direct addressing is not available to reference a data block. The only block that has a fixed place and that cannot be moved by garbage collection is the system block. But direct addressing does exist to read constants or to store code in the program memory.

Instruction set. The 136 instructions of Hip1 may be classified into 20 groups, as seen in Figure 4. Eight execution errors detected by the instructions' microcode are:

- OVERFLOW, for integer arithmetic operations;
- DIVZERO, division by zero;
- ERRBORNE, out-of-bounds error in a JUMP_INDEX instruction;
- VALIND, use of an uninitialized value (nil value);
- ADRIND, use of an uninitialized address (nil pointer);

• NODETACH, trying to activate a process that was not detached;
• ERRETURN, trying to return to the system process; and
• ERREX, trying to exchange with the system process.

If one of these eight error conditions is detected, the microcode stops the execution of the program, gives the control back to the system, and updates the program termination status and the system block. Program debugging is thus facilitated.

The instructions that modify the content of any register update the processor flags, allowing conditional jumps. The processor flags are:

• OVR, overflow;
• CARRY;
• NEG, sign bit;
• EQUAL, zero bit;
• LESST, less than bit; and
• GREAT, greater than bit.

Development tools. The very particular characteristics of a microprogrammed system ask for specialized tools to help the design and the debugging of such a system.[8] We wrote and tested the microprogram for the Hip1 processor thanks to the two following tools, which were designed by our staff:

• *A variable-definition microassembler.* In a first pass, the microassembler receives the microinstruction format definitions (widths, field places, default values) and the interpretation of symbolic names (opcodes, definitions for the microoperations and parameters). In the second pass it receives the microprogram written using the language defined during the first pass and outputs its translation in object code.

• *A simulator.* The simulator takes the assembled microcode and a description of the processor at the register-transfer level (list of the registers and description of their interconnections by a list of microoperations). It then uses this information to simulate the functional behavior of the microcode.

These tools are not dedicated and allow us to design microcode for virtually any processor. They are written in Pascal and used on VAX, HP9836, and Sun computers.

VLSI implementation

At the same time we were designing the bit-slice implementation, a part of our team worked on a VLSI implementation to practice and test new design methods for integrated circuits.[9]

Internal architecture. Basically, the VLSI implementation architecture of the processor is the same as the bit-slice implementation. But, in general, the different functional blocks have less possibilities than their bit-slice equivalent. For example, the microprogram sequencer allows only a single level of subprogram.

The microcode uses about 1300 words of 36 bits: longer and narrower than that used in the bit-slice version. This difference in size comes from the vertical characteristics of the microinstructions. The control microoperations are separated from the processing microoperations.

The execution of a microinstruction decomposes into four phases:

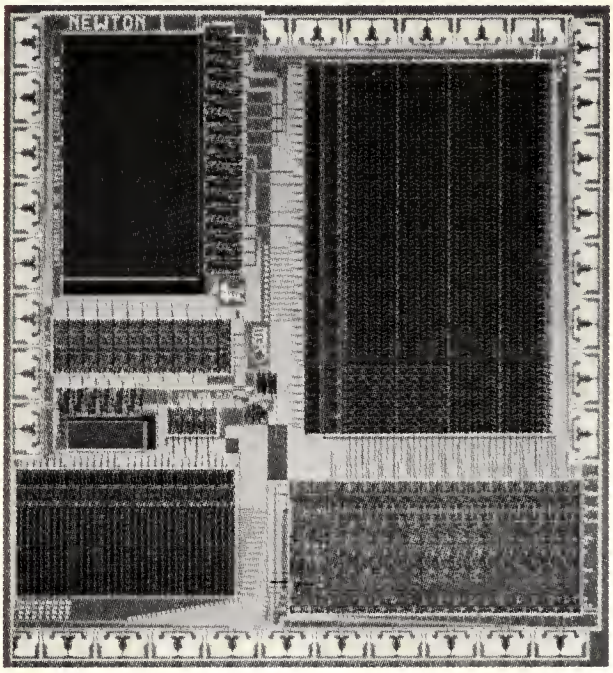
• loading of the micro-PC slave register; micro-ROM preload;
• micro-ROM reading;
• processing and loading of the data in the accumulator; and
• loading of the destination registers; loading of the micro-PC master register.

External architecture. From the user point of view, some differences exist between the two processors. The bit-slice version is a Harvard machine, with two separate memories for the data and programs; the VLSI version is a von Neumann machine with a single memory. Such a difference implies a slightly different heap management. In the VLSI version the programs are stored starting at address zero, toward the high end of memory. The first block of the heap lies at the high end of memory, and the heap grows toward the start of the memory. Except for some details, the garbage-collection algorithm stays the same.

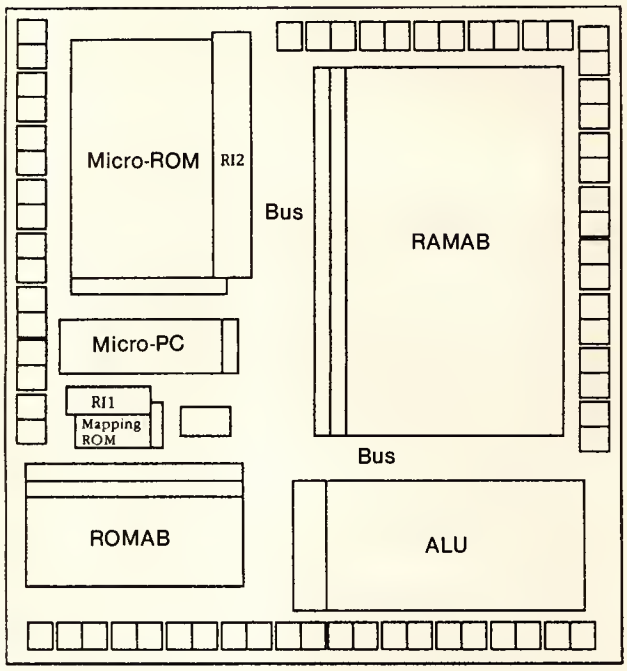
The VLSI version of the Hip processor also uses 16-bit words but has only eight registers of each type (byte, word, and heep), because of the space needed by these registers. They occupy about 20 percent of the chip surface.

Methodology. The circuit design follows a method developed at our school that avoids the need of a logical scheme.[1,10] The starting point is the Karnaugh table of each function. The sequential parts are realized asynchronously (without explicit memory elements) and may also be characterized by a Karnaugh table. By separately computing the 1 and 0 covering of the table, we obtain the network equation of the p and n transistors of a CMOS circuit. The covering of a Karnaugh table having up to 10 variables may be computed by a program.

We draw the layout on a symbolic grid consisting of a horizontal conductor (metal) and a vertical conductor (polysilicon). At the intersections may lie a crossing, a contact, or a transistor that has its gate bound to the vertical conductor. A program automatically translates this symbolic drawing into a



(a)



(b)

Figure 5. Chip microphotograph (a) and floor plan (b).

geometrical layout. Since the program takes care of the layout rules, designers do not need to worry about them. They can concentrate on placing the variables, and in doing that, take into account the connections to the neighboring cells.

This methodology allows a very short design time. The design and drawing effort of this processor may be estimated to be about eight man-months. The chip is implemented in 3-micron SACMOS technology (self-aligned CMOS).

Chip floor plan. Figure 5 shows the floor plan and the microphotograph of the chip. The block named ALU contains the entire processing part, except the register bank (RAMAB). The micro-PC block contains the control part, except the mapping-ROM and the micro-ROM. RI1 and RI2 are registers (12 and 36 bits) placed at the ROM output for testing purposes. They normally operate as parallel input-output registers but can be turned into serial mode. This capability allows us to examine or modify their contents. The ROMAB block contains constants (same bus address as RAMAB).

Here, we give some technical data to characterize the chip. It contains 71,551 transistors of which 46,944 are assigned to the micro-ROM, 14,576 to the register bank, and 3218 to the ALU. The chip surface occupies 20.1 mm^2. The bus and the contact pad take 8.72 mm^2 of this surface, the register bank, 4.65; the micro-ROM, 1.97, and the ALU, 1.94.

The goals set at the start of the project were wholly fulfilled.

1) For the first time in Switzerland, an integrated circuit for a 16-bit processor has been designed and realized.

2) A new course, named "Conception des processeurs" (processor design), benefits from this cooperation among software and hardware people. Students become acquainted with a global vision of computer architecture and its relations with high-level languages.

3) The compiler for the Newton language is not yet completed, and it is too early to give benchmark results. But the first tests of the processor in its bit-slice version are encouraging. As a matter of fact, a cross-compiler for the Pascal-S language,[11] a subset of Pascal, has been written on a Sun workstation, and the famous sieve of Eratosthenes can be executed in 6.3 seconds with a 5-MHz clock. (This program appears in the adjoining box.) The same program, compiled with Turbo Pascal, takes 5.1 seconds on a Macintosh Plus computer, which has an 8-MHz clock.

We have already started realizing the 32-bit versions of the processor, bit-slice as well as VLSI, and we foresee the first results early in 1988. We have also started the design of a processor adapted to the Prolog language, taking advantage of the same methodology and of our experience in this project.

The Sieve of Eratosthenes Program for Computing Prime Numbers

Here is the famous sieve of Eratosthenes program to compute prime numbers.

```
program primes;

const
  size = 8190;

var
  flags : array [0 .. size] of Boolean;
  i, prime, k, count, iter: integer;

begin
  writeln('10 Iterations');
  writeln('Hit return to start...');
  readln;
  for iter := 1 to 10 do begin
    count := 0;
    for i := 0 to size do
      flags[i] := true;
    for i := 0 to size do
      if flags[i] then begin
        prime := i + i + 3;
        k := i + prime;
        while k <= size do begin
          flags[k] := false;
          k := k + prime
        end;
        count := count + 1
      end
  end;
  writeln('There are ', count, ' primes.')
end.
```

And now, here is the result after grinding the above program with the Pascal-S compiler. The produced code has been commented, and blank lines have been inserted to make it clearer.

```
                ALLOCATE        #8204, #5               ; Program block allocation
                ENTER_SECT      primes$1                ; Go start the program

true$0          =               -1                      ; Program constants
size$1          =               8190
i$1             =               4+10+0                  ; Offsets of the program vars
prime$1         =               4+10+1
k$1             =               4+10+2
count$1         =               4+10+3
iter$1          =               4+10+4
flags$1         =               4+10+5


                .ENTRY          primes$1                ; Main program starts here

                ALLOCATE        #8, #2                  ; Procedure call to the
                ENTER_PROC      init_pascal_io          ; standard I/O initialization

                ALLOCATE        #13, #2                 ; writeln('10 iterations')
                LOAD_ADDRESS    s$0$strg, W15
                STORE_INDIR     W15, H0, #8
```

```
                ENTER_PROC      write_string
                ENTER_PROC      writeln

                ALLOCATE        #13, #2             ; writeln('Hit return to start...')

                LOAD_ADDRESS    s$1$strg, W15
                STORE_INDIR     W15, H0, #8
                ENTER_PROC      write_string
                ENTER_PROC      writeln

                ALLOCATE        #8, #2                      ; readln
                ENTER_PROC      readln

                SET_ONE         W15                         ; for iter := 1 to 10 do (1)
                STORE_INDIR     W15, GLOBAL, #iter$1
10:             -
                LOAD_INDIR      GLOBAL, #iter$1, W15
                COMPARE         W15, #10
                JUMP_GREAT      11

                SET_ZERO        W14                         ; count := 0
                STORE_INDIR     W14, GLOBAL, #count$1

                STORE_INDIR     W14, GLOBAL, #i$1           ; for i := 0 to size do (2)
12:             -
                LOAD_INDIR      GLOBAL, #i$1, W15
                COMPARE         W15, #8190
                JUMP_GREAT      13

                LOAD_DIRECT     #true$0, W14                ; flags[i] := true
                SET_REFER       GLOBAL, #4-flags$1, H4
                STORE_INDEX     W14, H4, W15

                INT_SUCC        W15, W15                    ; end of loop (2)
                STORE_INDIR     W15, GLOBAL, #i$1
                JUMP            12

13:             -
                SET_ZERO        W15                         ; for i := 0 to size do (3)
                STORE_INDIR     W15, GLOBAL, #i$1
14:             -
                LOAD_INDIR      GLOBAL, #i$1, W15
                COMPARE         W15, #8190
                JUMP_GREAT      15

                SET_REFER       GLOBAL, #4-flags$1, H4      ; if flags[i] then
                LOAD_INDEX      H4, W15, W14
                BIT             W14, #0
                JUMP_NOT_LESST  16

                INT_ADD         W15, W15, W14               ; prime := i + i + 3
                LOAD_DIRECT     #3, W13
                INT_ADD         W13, W14, W12
                STORE_INDIR     W12, GLOBAL, #prime$1

                INT_ADD         W15, W12, W14               ; k := i + prime
                STORE_INDIR     W14, GLOBAL, #k$1

17:             -                                           ; while k <= size do (4)
                LOAD_INDIR      GLOBAL, #k$1, W15
                COMPARE         W15, #8190
                JUMP_GREAT      18
```

```
                SET_ZERO        W14                         ; flags[k] := false
                SET_REFER       GLOBAL, #4-flags$1, H4
                STORE_INDEX     W14, H4, W15

                LOAD_INDIR      GLOBAL, #prime$1, W13       ; k := k + prime
                INT_ADD         W15, W13, W12

                STORE_INDIR     W12, GLOBAL, #k$1

                JUMP            17                          ; end of loop (4)
18:             -

                LOAD_INDIR      GLOBAL, #count$1, W15       ; count := count + 1
                INT_SUCC        W15, W14
                STORE_INDIR     W14, GLOBAL, #count$1

16:             -                                           ; end of if statement

                LOAD_INDIR      GLOBAL, #i$1, W15           ; end of loop (3)
                INT_SUCC        W15, W15
                STORE_INDIR     W15, GLOBAL, #i$1
                JUMP            14
15:             -

                LOAD_INDIR      GLOBAL, #iter$1, W15        ; end of loop (1)
                INT_SUCC        W15, W15
                STORE_INDIR     W15, GLOBAL, #iter$1
                JUMP            10
11:             -

                ALLOCATE        #13, #2                     ; write('There are ')
                LOAD_ADDRESS    s$2$strg, W15
                STORE_INDIR     W15, H0, #8
                ENTER_PROC      write_string

                LOAD_INDIR      GLOBAL, #count$1, W15       ; write(count)
                ALLOCATE        #19, #2
                STORE_INDIR     W15, H0, #8
                ENTER_PROC      write_integer

                ALLOCATE        #13, #2                     ; writeln(' primes.')
                LOAD_ADDRESS    s$3$strg, W14
                STORE_INDIR     W14, H0, #8
                ENTER_PROC      write_string
                ENTER_PROC      writeln

                END_PROG                                    ; end of program

s$3$strg: -                                                 ; Strings definitions
                .WORD                   ^A / p/, ^A /ri/, ^A /me/, ^A /s./, -
                                        0
s$2$strg: -
                .WORD                   ^A /Th/, ^A /er/, ^A /e /, ^A /ar/, -
                                        ^A /e /, 0
s$1$strg: -
                .WORD                   ^A /Hi/, ^A /t /, ^A /re/, ^A /tu/, -
                                        ^A /rn/, ^A / t/, ^A /o /, ^A /st/, -
                                        ^A /ar/, ^A /t./, ^A /../, 0
s$0$strg: -
                .WORD                   ^A /10/, ^A / I/, ^A /te/, ^A /ra/, -
                                        ^A /ti/, ^A /on/, ^A /s/@8
```

Acknowledgments

The Hip processor is the fruit of the cooperation of two laboratories, one dedicated to hardware and the other to software. The following people, staff of one of these laboratories, also participated in this project: Y. Abdoun (VME interfacing), C. Bernard (printed circuits), and S. Mourtada (software). The Centre suisse pour l'electronique et la microtechnique (CSEM, Neuchatel-Switzerland) lent us its equipment to design the final layout of the integrated circuit. The Commission pour l'encouragement de la recherche scientifique and the Fonds national suisse de la recherche scientifique (Switzerland) financed a part of the project.

References

1. C. Piguet, J. Zahnd, A. Stauffer, M. Bertarionne, "A Metal-Oriented Layout Structure for CMOS Logic," *IEEE J. Solid-State Circuits,* June 1984, pp. 425-436.

2. C. Rapin and J. Menu, "The Newton Language," *SIGPLAN Notices,* Aug. 1981, pp. 31-40.

3. J. Menu, "The General Heap, A High-Level Concept," PhD thesis 495, Ecole polytechnique federale, Lausanne, Switzerland, 1983.

4. A.S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM,* Mar. 1978, pp. 237-246.

5. D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms,* Addison-Wesley, Reading, Mass., 1973.

6. J. Cohen and A. Nicolau, "Comparison of Compacting Algorithms for Garbage Collection," *ACM Trans. on Programming Languages and Systems,* Oct. 1983, pp. 532-553.

7. E. Sanchez, "Processeur adapte au langage Newton: version circuit en tranches," internal report, Laboratoire de systemes logiques, Ecole polytechnique federale, Lausanne, 1986.

8. E. Sanchez and P. Sommer, "Outils d'aide a la conception des systemes microprogrammes," internal report, Laboratoire de systemes logiques, Ecole polytechnique federale, 1985.

9. P. Sommer, "Processeur adapte au langage Newton: version circuit integre," internal report, Laboratoire de systemes logiques, Ecole polytechnique federale, 1986.

10. A. Stauffer and J. Zahnd, "Methodes de synthese des systemes logiques CMOS," bulletin ASE/UCS, Zurich, Jan. 1984, pp. 18-34.

11. N. Wirth, "Pascal-S: A Subset and Its Implementation," in D.W. Barron, *Pascal—The Language and Its Implementation,* John Wiley and Sons, New York, 1981.

Eduardo Sanchez is engaged in teaching and research in the areas of logical design and computer architecture at the Ecòle polytechnique federale in Lausanne, where he has been working since 1977. His current interests include compilers, microprogramming, logic programming, and Prolog-oriented computer architecture.

Sanchez received the diploma of electrical engineer from the Universidad del Valle in Cali, Colombia, in 1975 and his PhD from the Ecole polytechnique federale in 1985.

Patrick Sommer works on an Ecole project of automatic design methods for CMOS logic circuits with ordered layout. His interests lie in VLSI design and compilers.

Sommer received the diploma of electrical engineer from the Ecole polytechnique federale in 1982.

Jacques Menu lectures at the University of Geneva and is a scientific associate at CERN (the Centre europeen de recherches nucleaires) in Geneva, Switzerland. His current interests include logic programming, specification languages, and rewrite rules as a problem-solving paradigm.

Menu received his PhD from the Ecole polytechnique federale in 1983.

Christian Iseli is a diploma candidate at the Ecole polytechnique federale and is currently working on the design of a small general-purpose microprocessor to be used in watches. His research interests lie in high-level-language oriented architecture, compilers, and operating systems.

Questions concerning this article may be directed to the authors at Ecole polytechnique federale de Lausanne, 16. ch. de Bellerive, CH-1007 Lausanne, Switzerland.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 165 Medium 166 High 167

FEATURE

A Fast Integer Binary Logarithm of Large Arguments

Reinhard Maenner
Physical Institute,
University of Heidelberg

The logarithm is a function defined on the space of nonnegative real numbers ($x \in \mathcal{R}$, $x > 0$). Its value is real ($x \in \mathcal{R}$, $-\infty < x < \infty$) and can be interpreted as giving the order of magnitude of the argument. We often therefore use logarithms in which variables with a large range of values are to be handled. A typical example is the processing of acoustic signals. The human ear has a logarithmic sensitivity over a range of $1{:}10^6$. An analysis example of individual Fourier components, that is, of their time dependencies, is meaningful only in considering the logarithms of these components.

Traditionally, we have executed such computations with floating-point arithmetic. All mainframes and most minicomputers use hardware floating-point processors for this task. High-end microprocessors typically exploit floating-point coprocessors to support real arithmetic. Low-cost systems, however, still emulate floating-point operations by software. Typical examples of low-cost systems are personal computers such as the Macintosh Plus or the Atari 1040 ST. Such systems are increasingly applied to control small experiments because of their low-cost graphic capabilities and good human interfaces. Here the emulation of floating-point operations and, among others, the computation of logarithms generally degrades the system throughput by two orders of magnitude.

In some cases this performance degradation can be avoided. If the dynamic range of the parameters handled can still be represented with integer numbers, a floating-point computation is not necessary. A 32-bit microprocessor usually can handle integer numbers of up to 64 bits (32 bit * 32 bit multiply), corresponding to a dynamic range of $1{:}10^{19}$. Thirty-two-bit integers still cover a dynamic range of $1{:}10^9$.

Fortunately, the logarithm is one of the real functions that can be approximated quickly by integer and logical operations. Here, I review methods to compute such approximations and propose a new method that can be implemented easily in software only, is computed very fast, and allows users to choose an optimum regarding required table space and approximation error.

Simple and fast ($10\,\mu$), this new procedure requires no hardware and features a 10^{-6} approximation error.

Previous work

In the following we will consider only integer arguments with a reasonable number of bits. If this number is too low, for example, ≤ 16, the computation of the logarithm becomes trivial, for example, using a lookup table (as discussed later). If the number of bits is too high, say, ≥ 64, the computation has no practical importance. This is true because even high-performance processors at most handle 64-bit integers (usually results of a 32-bit * 32-bit multiply) and because larger integers cover such an immense dynamic range that cannot be used in most computations.

Lookup tables using ROMs or RAMs. Because we consider only a finite set of arguments, we can in

0272-1732/87/1000-0041$01.00 © 1987 IEEE

principle compute the logarithm of each possible argument in advance and store all results in a table. A lookup table can then be used to "compute" the logarithm extremely fast. The computation time here equals the access time of read-only memories (ROMs) or random access memories (RAMs) storing the table, which is typically 50-200 ns. This method, proposed by Brubaker and Becker,[1] can however only be used with a rather limited range of arguments. Because for n-bit integers, one table entry is required for each one of the 2^n possible arguments, the necessary ROM/RAM space grows exponentially with the argument word length. Even by providing huge tables of megabyte size (the typical main memory size of today's personal computers), the argument length would be limited to a little above 20 bits.

Linear approximation. For larger arguments, some kind of computation has to be done in real time. Note that for some special arguments this computation is trivial. The binary logarithm (ld) is defined by $x = 2^{ld(x)}$. The argument is represented in binary form by

$$x = \sum_{i=0}^{n} a_i 2^i, \tag{1}$$

where $a_i \in \{0,1\}$. If therefore only a single one of the a_i's is set, say a_k, $x = 2^k$ and $ld(x) = k$, giving an integer result. Note that in this case the result equals the number of the single bit set. In all other cases, that is, $2^k < x < 2^{k+1}$, the bits additionally set represent the binary fraction of the argument. This can be seen by writing

$$\begin{aligned} x &= \sum_{i=0}^{k} a_i 2^i = 2^k + \sum_{i=1}^{k} a_{k-i} 2^{k-i} \\ &= 2^k + 2^k \cdot \sum_{i=1}^{k} a_{k-i} 2^{-i} \\ &= 2^k \cdot \left[1 + \sum_{i=1}^{k} a_{k-i} 2^{-i}\right] = 2^k \cdot (1+z). \end{aligned} \tag{2}$$

Here again, k is the number of the uppermost bit set. The binary fraction z is always in the range $0 \leq z < 1$.

Taking the logarithm of the last expression yields $ld(x) = k + ld(1 + z)$, so the $ld(1 + z)$ also lies within the range $0 \leq ld(1 + z) < 1$. One method to compute the logarithm therefore requires

- splitting the argument into the uppermost bit set and the remainder,
- determining the number of the uppermost bit set,
- interpreting the remainder as a binary fraction with bit numbering relative to the uppermost bit,
- computing an approximate logarithm $ald(z) \cong ld(1 + z)$, which maps the range $0 < z < 1$ onto itself, and
- adding up the number of the uppermost bit set and the result of this mapping.

It should be noted that for an argument range of, say 64 bits, a result range of only 6 bits would be required for true integer result values. If computed in this way, the relative result error would be between $1/64 \cong 2$ percent and $1/1 = 100$ percent. A much higher precision is achieved if the result is scaled before truncation. This can be done by multiplying the result by a constant to exploit the full available integer width. For 64 bits the constant would be $2^{64 - ld(64)} = 2^{58}$. This multiplication can be done by shifting the result left an appropriate number of bits.

Mitchell[2] gave the simplest method for computing an approximate value of $ld(1 + z)$. He proposed using the approximation

$$\begin{aligned} ald(x) &= ald\left[2^k \cdot (1+z)\right] \\ &= k + ald(1+z) = k + z + f(z). \end{aligned} \tag{3}$$

This approximation corresponds to a linear interpolation between $ld(2^k) = k$ and $ld(2^{k+1}) = k + 1$. The computation requires only a normalization of the argument. If the uppermost bit set is shifted to the carry, the argument bit width minus the shift count equals k, and the remainder can just be added to the scaled value of k. This can be seen in the example shown in Figure 1 (a small bit width is taken for convenience).

The maximum error in this approximation is found at $z = 0.443$ and is equal to 0.087, that is, roughly 10 percent. For many applications this error is still acceptable. There, the simplicity of the described algorithm allows a very fast implementation in software. The routine displayed in Figure 2 computes $ald(x)$. The routine is written for a 68000 microprocessor (both personal computers mentioned above use this CPU) and assumes 32-bit, nonnegative integers for arguments. Results also have this range.

When this routine is executed on a 16-MHz CPU, it requires an execution time of maximally 43.6 μs (argument = 1) and minimally 6.1 μs (argument $\geq 2^{30}$). If the arguments are distributed randomly over the argument range, the routine requires an average computing time of 7 μs.

Due to the simplicity of Mitchell's approximation, it can easily be implemented in hardware. Jankowski-Tebe[3] described a logarithmic counter operating according to this method.

The approximation error can be reduced in a straightforward manner in two ways. Because with

Mitchell's method $ald(x)$ is always smaller than $ld(x)$, the maximum error can be reduced roughly by a factor of 2. This is done by adding a positive constant chosen to minimize $| ld(1 + z) - z |$ in the interval $z \in [0,1]$. Additionally, it is possible to make a linear approximation between all argument values 2^i as supporting points. But one possible improvement would be to use more supporting points (between the values 2^i) and to make a linear approximation between them. Combet et al.[4] and Hall et al.[5] suggested using the approximation $ald(1 + z) = az + bf(z) + c$ with proper values for a,b,c and a proper function f. Both methods reduce the maximum error of 0.086 (Mitchell); Combet's method reduces to a value of 0.013 and Hall's method, to a value of 0.017. We pay for the reduction with higher computational and/or hardware effort(s). Besides requiring at least four additions, a multiplication is also required, which takes about 30 times longer than a register addition (for a 68000 CPU).

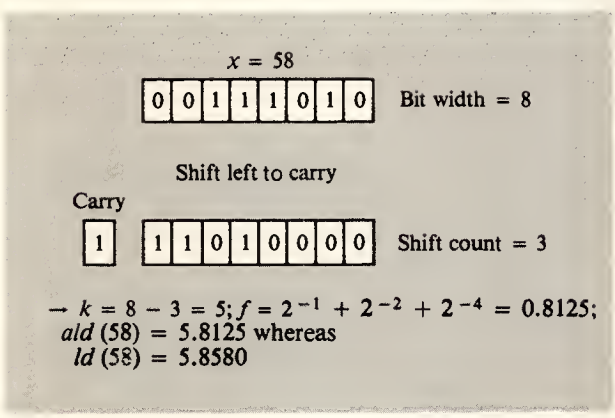


Figure 1. Principle of Mitchell's approximation.

Nonlinear approximation. The approximation error can clearly be reduced further if we use a nonlinear approximation. Marino[6] proposed a quadratic approximation that can be implemented in hardware. This method yields a maximum error of 0.004, which is smaller than Mitchell's error by a factor of 20. However, the usage of special-purpose hardware for the implementation of the logarithm is often not feasible, for example, for the personal computers mentioned earlier. A realization by software requires considerably more computational effort, even when compared to Combet's and Hall's approximations.

Lookup table with differential group PLAs. There are trade-offs between computational speed and approximation errors on one hand and between implementation in hard/software and the cost/performance ratio on the other hand. Lo and Aoki[7] proposed implementing the logarithm by special-purpose hardware using a programmable logic array (PLA) plus standard electronic devices such as adders. They first use the approximation

$$\begin{aligned} ald\,(x) &= ald\,[2^k \cdot (1+z)\,] \\ &= k + ald(1+z) = k + z + f\,(z), \end{aligned} \tag{4}$$

where $f(z)$ is an approximation to $ld(1 + z) - z$. This method was chosen, because $f(z)$ often has identical values for certain argument ranges. The idea is to group all argument values belonging to the same value of $ld(1 + z) - z$ and to assign each group to one product term of the PLA. However, the number of groups required is still comparable to the number of possible arguments. For a binary fraction with a width of 8 bits, 107 PLA groups are obtained instead of 256 groups using ROMs or RAMs. The maximum

```
* This procedure computes x = log2 (x) * 2**26
* using Mitchell's method
*
* Entry conditions:        D0.L = argument (>0)
* Return conditions:       D0.L = result
* D1 will be destroyed
*
LOG2:  MOVE.W    #31, D1     ; preset maximum loop counter
LOG2A: LSL.L     #1,D0       ; normalize argument by shifting
       DBCS      D1,LOG2A    ; uppermost bit to carry
       AND.B     #$C0,D0     ; clear lowest six bits
       OR.B      D1,D0       ; insert there number of upper-
                             ; most bit set
       ROR.L     #6,D0       ;rotate it to uppermost five bits
       RTS                   ; return
```

Figure 2. Routine for computing $ald(x)$ using Mitchell's approximation.

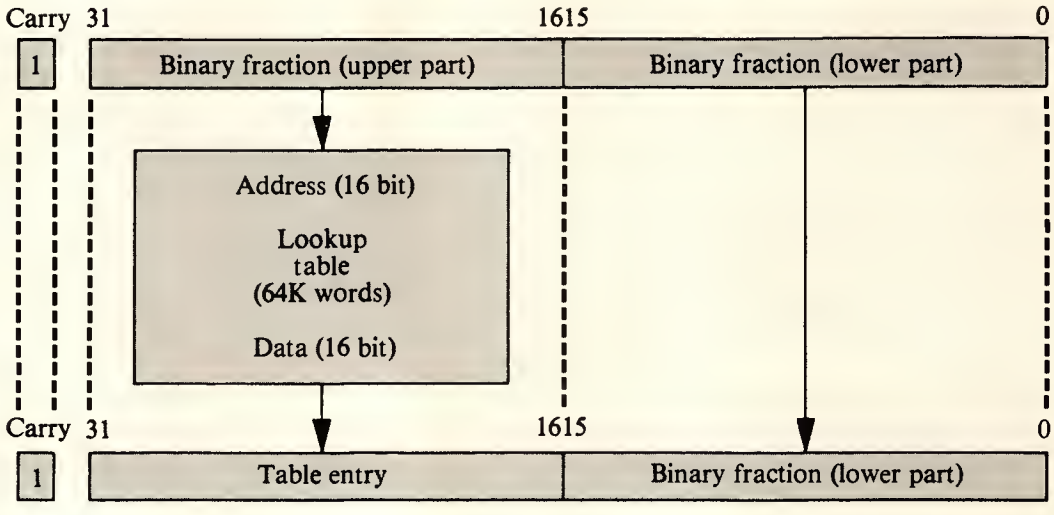


Figure 3. Principle of proposed approximation.

```
* This procedure computes x = log2(x) * 2**26
* using Mitchell's method combined with table lookup
*
* Entry conditions:      D0.L = argument (>0)
* Return conditions:     D0.L = result
* D1, D2 and A0 will be destroyed
LOG2:   LEA     Table, A0       ; setup pointer to lookup table
        MOVE.W  #31, D1         ; preset maximum loop counter
LOG2A:  LSL.L   #1, D0          ; normalize argument by shifting
        DBCS    D1,LOG2A        ;  uppermost bit to carry
        SWAP    D0              ; get high word of binary fraction
        CLR.L   D2              ; initialize high word to zero
        MOVE.W  D0,D2           ; copy high word of binary fraction
        ASL.L   #1,D2           ; make word offset for table access
        MOVE.W  0(A0,D2.L), D0; replace 1st 16 bit of binary
        SWAP    D0              ;  fraction with lookup table entry
        AND.B   #$C0,D0         ; clear lowest six bits
        OR.B    D1,D0           ; insert there number of upper-
                                ;  most bit set
        ROR.L   #6,D0           ; rotate it to uppermost five bits
        RTS                     ; return
```

Figure 4. Routine for computing $ald(x)$ using proposed approximation.

error in this example is 0.017, roughly the same as with the methods of Combet and Hall.

Combination method

I propose another combination of Mitchell's method and a lookup table. It can be implemented completely in software and is therefore well suited for usage in low-end personal computers. The basic idea is to split the binary fraction z into

$$z = \sum_{i=1}^{k} a_{k-i}2^{-i} = \sum_{i=1}^{m} a_{k-i}2^{-i} + \sum_{i=m+1}^{k} a_{k-i}2^{-i}, \quad (5)$$

where m is the number of bits used for addressing a lookup table. The approximation is then taken as

$$ald\ (1+z) = f(a_{k-1}, \ldots, a_{k-m}) + \sum_{i=m+1}^{k} a_{k-i}2^{-i}. \quad (6)$$

The function f is given as a lookup table. For all combinations of $a_{k-1}, \ldots, a_{k-m}$, the table contains one entry each with a value precomputed to minimize the approximation error. The operations required are

- normalizing the argument as done in Mitchell's method and
- replacing the first m bits of the binary fraction by an element of the lookup table.

Figure 3 shows operations for the case $m = 16$.

The value of m can be chosen arbitrarily. In principle, m might be anything from zero (Mitchell's method) to k (Brubaker's and Becker's method). This means that m can be optimized in regard to table size (2^m words, word width = m bits) and approximation error. In practice, one would choose m according to the available access width of the memory used, usually 8 or 16 bits.

It is easy to estimate the error made by this approximation. Without a lookup table, the result is the same as with Mitchell's algorithm. Using the first m bits of the binary fraction for a lookup table replaces these m bits by their exact, precomputed values. The only possible bit errors are in the bit positions for $m+1$ down to zero. If these bits were set to zero, the error would be $2^{-(m+1)}$. However, these bits are set according to the linear approximation $ald(1 + z) = z$. They therefore interpolate linearly between the exact values taken from the table. The error for these bits is therefore exactly the same as Mitchell's error scaled down by a factor of 2^{-m}. In the example given earlier, we have $n = 32$ and $m = 16$. The total error therefore equals $0.087 * 2^{-16} = 1.3 * 10^{-6}$. The routine shown in Figure 4 computes the binary logarithm in the proposed way.

When this routine is executed on a 16-MHz CPU, it requires an execution time of maximally 46.6 μs (argument = 1) and minimally 9.1 μs (argument $\geq 2^{30}$). If the arguments are distributed randomly over the argument range, the routine requires an average computing time of 10 μs.

The proposed algorithm to compute the binary logarithm has advantages when compared with earlier methods. Whereas some methods use special-purpose hardware to speed up computation or to reduce storage requirements, the new method can be implemented easily in software only. This capability is important, if a logarithm function is required in systems that do not use floating-point hardware and if special-purpose hardware add-ons are not feasible as with low-end personal computers.

I presented a procedure for a 68000 microprocessor, which is used in popular systems like the Macintosh and the Atari 1040 ST. This procedure is very simple – it uses 14 machine instructions only – and is computed quickly. For argument values distributed randomly over a 32-bit range, the average computation time is on the order of 10 μs for a 16-MHz processor, comparable to the speed of a floating-point coprocessor. Moreover, the approximation error is very small, depending on the actual implementation. For a 64K-word lookup table, this error is on the order of magnitude of 10^{-6} for the given example, a factor of 100 to 1000 smaller than the errors obtained with earlier approximations.

References

1. T.A. Brubaker and J.C. Becker, "Multiplication Using Logarithms Implemented with Read-Only Memory," *IEEE Trans. Comp.*, 1975, pp. 761-766.
2. J.N. Mitchell, Jr., "Computer Multiplication and Division Using Binary Logarithm," *IEEE Trans. Electr. Comp.*, 1962, pp. 512-517.
3. K. Jankowski-Tebe, "Logarithmus dualis fur Digitalsignale," *Elektronik*, 1983, pp. 99-100.
4. M. Combet, H. van Zonneveld, and L. Verbeck, "Computation of the Base Two Logarithm of Binary Numbers," *IEEE Trans. Elect. Comp.*, 1965, pp. 863-867.
5. E.L. Hall, D.D. Lynch, and S.J. Dwyer III, "Generation of Products and Quotients Using Approximate Binary Logarithm for Digital Filtering Applications," *IEEE Trans. Comp.*, 1970, pp. 97-105.
6. D. Marino, "New Algorithms for the Approximate Evaluation in Hardware of Binary Logarithm and Elementary Functions," *IEEE Trans. Comp.*, 1972, pp. 1416-1421.
7. H.-Y. Lo and Y. Aoki, "Generation of a Precise Binary Logarithm with Difference Grouping Programmable Logic Array," *IEEE Trans. Comp.*, 1985, pp. 681-691.

Reinhard Maenner joined the Physical Institute of the University of Heidelberg, West Germany, in 1975, after earning a Diplom in physics from the University of Munich. At the Institute, he developed a data acquisition system for large-scale experiments in nuclear physics. As part of this work, he wrote a real-time, demand-paged, virtual-memory multitasking operating system for a PDP-11/45. For these efforts he received the doctoral degree in 1979.

The author of over 25 papers on nuclear physics, computer architecture, and image processing, Maenner currently heads the group at the Physical Institute developing the Polyp multiprocessor. He is also a cofounder of Heidelberg Instruments, a company producing computerized optical instruments. He is a member of the ACM.

Questions about this article can be directed to the author at Physical Institute, University of Heidelberg, Philosophenweg 12, D-6900 Heidelberg, West Germany.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 153 Medium 154 High 155

FEATURE

Effective Implementation of a Parallel Language on a Multiprocessor

Thomas L. Sterling, Albert J. Musciano, Ellery Y. Chan, and Douglas A. Thomae
Harris Corporation

A new multiprocessor execution environment integrates semantics of parallel control with mechanisms for synchronization of concurrent tasks.

Present technology provides computer designers with the potential for assembling a powerful computer at low cost by using multiple microprocessors to execute a single program. Such a machine is a member of the class of multiple-instruction stream, multiple-data stream (MIMD)[1] parallel computers called multiprocessors. These machines are distinguished by a number of roughly equivalent, tightly coupled processors.

Researchers have implemented several experimental multiprocessors, such as the CM* machine.[2] Unfortunately, more than a decade of active experimentation with single-program execution on multiprocessor architectures has produced few effective methods for achieving substantial performance gains. As a result, multiprocessors incorporating more than four processing elements have had little impact on commercial and scientific computing. We need advances in parallel-programming models and execution techniques to mate the abundance of inexpensive hardware to parallel applications.

A multiprocessor-based, parallel-program execution environment called SPOC (Simultaneous Pascal on Concert) provides a solution to the resource-utilization problem for a broad range of applications using small- and medium-scale multiprocessors. SPOC is based on a *concurrent thread* model of parallel computation. Each thread is a sequence of instructions that can be independently scheduled for execution.

Simultaneous Pascal[3] is a high-level language that provides the constructs to explicitly delineate and organize the concurrent threads of an application program. The *rendezvous control mechanism* (RCM) is the runtime strategy employed by SPOC to synchronize the termination of multiple active threads as an effective solution to the "join problem."

The Concert[4] multiprocessor testbed developed at MIT and Harris Corporation is the physical parallel-execution medium for programs written in Simultaneous Pascal (see Figure 1).

The parallel semantics of Simultaneous Pascal and the RCM were developed together, and each reflects the power and limitations of the other. SPOC is the synthesis of these two methods with the parallel resources of the Concert multiprocessor. It provides a complete execution environment for user applications.

Multiprocessor characteristics

From a physical standpoint, we can view a multiprocessor in the following terms:

- Number of processors,
- Processor power in terms of speed and word width,
- Memory size, speed, and distribution, and
- Communication interconnection for processors and memories.

0272-1732/87/1000-0046$01.00 © 1987 IEEE

This view is limited, since it conveys only the physical structure of a machine. Although we can deduce the peak performance level of the multiprocessor, this view gives no indication of the logical machine residing above the hardware, or its actual behavior.

The abstract model of a multiprocessor presents the machine as a collection of primitive functions that manage, process, and coordinate concurrent activities. We can characterize such a model in the following terms:

- Types of parallelism that can be exploited within an algorithm,
- Synchronization among concurrent activities,
- Context initialization and switching,
- Scheduling strategies for concurrent activities, and
- Locality of distributed objects and relevant activities.

The characteristics of a machine determine the *granularity* of the parallelism that can be used effectively by that machine. Granularity is the smallest amount of work that can be scheduled for execution without incurring significant performance losses due to overhead. We define overhead as the work performed by a multiprocessor to manage the execution of parallel activities that would not be performed by a uniprocessor executing the same program. Examples of overhead are synchronization of tasks, task scheduling, and context switching. A meaningful measure of the minimum-task granularity that can be efficiently employed is the amount of overhead incurred in support of the task.

Scalability is the maximum number of processors that can be efficiently used by a multiprocessor. This limit is largely determined by the system's granularity and performance degradation due to contention for access to shared resources. Scalability determines the most powerful multiprocessor that could be implemented with a particular architectural approach. The scalability of the architecture, combined with the parallel nature of a particular application program, defines the upper bound of the speed with which the program can be executed.

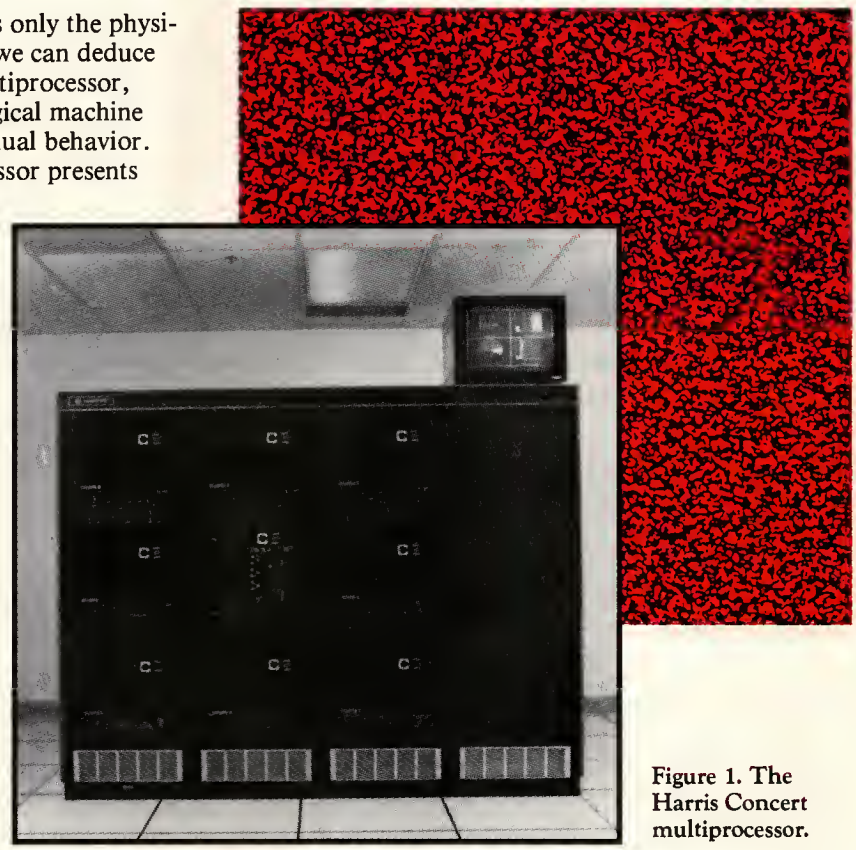


Figure 1. The Harris Concert multiprocessor.

Models of parallel synchronization

Several models of parallel computing exist, each imposing its unique synchronization requirements. Such models include multiprogramming, communicating sequential processes, and data flow.

Multiprogramming systems are very coarsely grained, running a number of programs at one time and using only the parallelism at the job boundaries. Although the multiprogramming approach exhibits limited scalability, Wulf and Bell employed it in the experimental C.mmp,[5] while Matelan used it in the commercial Flex/32.[6]

The Communicating Sequential Processes (CSP)[7] model is coarsely grained, with the available parallelism defined at the process level. Data passes between pairs of processes, either of which may be suspended until the other is ready to communicate. Here, too, scalability is limited, and the class of ap-

plications that may be readily captured with this model is restricted. To assist in this latter problem, researchers have developed several high-level languages that reflect this model, including Ada,[8] Occam,[9] and Concurrent Pascal.[10]

The data-flow computing model[11] uses fine-grained parallelism with data-driven synchronization. Its value-oriented paradigm limits the types of problems to which it is applicable; its fine-grainedness makes achieving an efficient multiprocessor implementation difficult.

Philosophy of parallel languages

Great debate surrounds the problem of writing parallel programs. Some parallel-programming languages require programmers to explicitly indicate the parallelism in their programs. Others require programmers to divide a program among several processors, and to provide synchronization and communication mechanisms. At the other end of the spectrum, smart compilers are used to derive the parallelism in a sequential program.

Many languages fall between these two extremes. A class of several languages reveals parallelism without programmer intervention. Unfortunately, these languages, such as data flow[12] and graph reduction,[13] require a purely value-oriented programming style and restrict the class of usable applications.

statement ::= ...
forall_statement
fork_statement
using_statement
lock_statement

forall_statement ::= **forall** *id* := *expr* **to** *expr* **do**
statement

fork_statement ::= **fork** *statement_list* **join**

using_statement ::= **using** *decl_list* **do**
statement

decl_list ::= *declaration* [; *decl_list*]

declaration ::= *id_list* : *type_id*

id_list ::= *id* [, *id_list*]

lock_statement ::= **locking** *expr* **do**
statement

Figure 2. Simultaneous Pascal parallel constructs.

The SPOC philosophy is to adopt a suitable sequential language and augment it with explicit parallel constructs. This philosophy encourages programmers to be conscious of the parallelism in their programs and helps them to write naturally parallel algorithms. Although this approach requires some initial learning, programmers quickly adapt to the new language features.

The SPOC model does not demand that programmers become intimately familiar with the underlying parallel machine. The burden of effectively mapping a parallel program onto the machine is left to the compiler and runtime system. In addition to freeing the programmer from this arduous task, the SPOC model promotes the creation of portable parallel code.

Simultaneous Pascal

The Simultaneous Pascal programming language is a superset of Standard Pascal.[14] Simultaneous Pascal contains all language features found in Standard Pascal and augments these features with several parallel-control constructs. These constructs allow the programmer to specify the portions of the program that can execute in parallel, to provide for a finer grained control of variable scoping, and to allow controlled access to global objects.

Simultaneous Pascal, unlike other parallel versions of Pascal such as Concurrent Pascal, promotes a thread-based model of parallel control. We can view a program as a collection of sequential pieces of code, or *threads*. Any single thread, once scheduled to execute upon a processor, will run to completion without interaction from any other processor. When a thread has finished executing, subsequent threads are then made available for execution. Parallel constructs in Simultaneous Pascal allow a programmer to indicate the threads in a program and to delineate the precedence constraints between various threads. The syntax of these new constructs is shown in Figure 2.

Three principal types of parallelism are available to the programmer in Simultaneous Pascal. Homogeneous parallelism allows the programmer to specify a single statement, many copies of which are to be executed in parallel. In heterogeneous parallelism, the programmer specifies several different statements to be executed in parallel. Operator-level parallelism allows the operands (which may be expressions) of an operator to be evaluated in parallel.

The Forall statement indicates homogeneous parallelism. The Forall statement allows the programmer to specify an index variable, a range of values over which the variable is to be instantiated, and a statement that is to be associated with each instance of the index variable. A simple Forall statement would be:

```
forall i := 1 to max do
  a[i] := b[i] + c[i];
```

In this example, *i* is the index variable, which will assume values over the range 1 to max. The statement

```
a[i] := b[i] + c[i]
```

will be executed max times in parallel, with each instance of the statement associated with a different value of *i*. Thus, parallel vector addition is being performed. All of the instances of the body of the Forall statement will finish executing before the statement after the Forall is scheduled for execution.

Heterogeneous parallelism is implemented with the Fork statement. The Fork statement allows the programmer to specify a list of statements, all of which may be executed in parallel. A simple Fork statement would be:

```
fork
  a := 1;
  b := 2;
  c := 3
join
```

The assignment statements may be executed concurrently. When all statements have completed, the statement following Join is scheduled for execution.

Operator-level parallelism allows the programmer to indicate those portions of an expression that can be evaluated in parallel. The operands of any binary operator can be executed in parallel by enclosing that operator within vertical bars. The following example uses operator-level parallelism:

```
r := sqr(sin(x))|*|sqr(cos(x));
```

Since calculating the square of the sine or cosine of a value is a relatively long process, the two subexpressions can be evaluated in parallel, reducing the time needed to evaluate the right-hand side of the Assignment statement. When both subexpressions have been evaluated, their product will be computed and assigned to *r*.

The lowest resolution of scoping in Standard Pascal is the procedure or function. That is, the *scope* of a procedure or function is the smallest portion of a program in which a name can be known. When writing parallel programs, the need often arises to have variables whose scope is local to the current thread. To meet this need, Simultaneous Pascal provides a new construct, Using, that allows finer grained scope control down to the level of a single statement.

The Using construct allows the programmer to specify a list of variables and a statement. The variables will only be known within the statement, and storage for those variables will only be allocated while the statement is executing. For example, the following code reverses the order of the elements of an array:

```
forall i := 1 to max div 2 do
  using temp : integer do
    begin
    temp := a[i];
    a[i] := a[max - i + 1];
    a[max - i + 1] := temp
    end
```

Each instance of the body of the Forall statement will be associated with a variable named *temp*. The storage allocated for temp will be local to each thread. Thus, each thread can assign to and use temp without interfering with other threads as they use their private temp variables. Without scope resolution at the statement level, note that the programmer would be required to create a temporary array so that each thread would have a unique location to use for intermediate storage. The naive approach, which is to simply declare temp in the scope containing the Forall statement, is incorrect; all threads will assign to and attempt to use the single instance of temp simultaneously, yielding incorrect results.

Simultaneous Pascal promotes a thread-based programming model. Unlike the CSP model of parallel processing, no way exists for a thread to send messages to other threads. However, the problem of having exclusive use of a global-data object is not eliminated. While CSP models would place the global object in some sort of monitor, and use semaphores to control entry into the monitor, Simultaneous Pascal does not have the ability to suspend threads waiting for access to the monitor.

Simultaneous Pascal does give the programmer the ability to serialize otherwise-parallel access to some global object by means of the Locking statement. The Locking statement allows the programmer to specify a lock variable and a statement to be executed if the lock is unlocked. The following code fragment shows how a thread would access the registers of some hardware device, guaranteeing that it has exclusive use of the device:

```
locking device.lock do
  begin
  device.control := write_data;
  device.data := 0;
  end
```

The variable named *device.lock* must be of type lock, which is unique to Simultaneous Pascal. Lock may be used like any other simple type, except that variables of type lock can only be passed as var parameters or used as the object of a Locking statement. Variables of type lock have only two values: locked and unlocked. Initially, a lock is set to the unlocked state. When the Locking statement is encountered, the thread will loop until the specified lock is in the unlocked state. The lock will then be placed into the locked state, and the thread will con-

tinue with the execution of the body of the Locking statement. When the body of the Locking statement has finished, the lock is returned to the unlocked state.

Note that threads *do not* suspend themselves when a locked lock is encountered. The processor on which the thread is executing will poll, waiting for the lock to become unlocked. Misusing locks will impair parallel-processor performance.

Runtime support mechanisms

SPOC presents many interesting implementation problems. The two most important involve (a) synchronizing several concurrent threads and (b) managing multiprocessor memory.

Rendezvous control mechanisms. A thread can be scheduled for execution when a specified set of preceding threads has run to completion. The difficulty of determining when all preceding threads have terminated is known as the *join problem*. Various computing models approach the join problem differently. These approaches can be distinguished by the amount of knowledge the system has concerning the location at which each join is to be performed. This *join location* (or *join point*) is a data structure in memory where the synchronization of multiple tasks is controlled.

Solutions to the join problem. At one extreme, special-purpose, synchronous systems such as systolic arrays require no join point, since hardware timing guarantees that join conditions will be satisfied when the operation is performed. At the other extreme, dynamic data-flow models require associative search techniques to match tokens destined for the same operation. The join location is not known until one of the tasks (in this case a single operation) has completed. Since the other task required for synchronization cannot know this location, a search must be made to locate the appropriate join point.

Some variants of the CSP model and the static data-flow model specify the join location at compile time. In the case of CSP, semaphores are defined to control the handshaking between concurrent processes. For static data flow, activity templates are defined at compile time that buffer and synchronize arriving values to determine when that template's operation is ready to be executed. There is no runtime cost in determining the join location, but the static program flow necessitated by the compile-time join locations can restrict the usefulness of these models.

SPOC uses the RCM to solve the join problem. (Our use of "rendezvous" bears no relationship to the concept of rendezvous in the Ada programming language.) The RCM allows SPOC to remember the join point of a thread when that thread is created. This method provides flexibility in programming, while eliminating the need to search for join points at runtime. We refer to this kind of join point determination as *runtime preassignment*.

In the Fork statement, the threads representing the various statements must join at the join keyword before program flow can exit the Fork statement. The threads created by the Forall statement must join before control passes on to the next statement. Thus, the parallel semantics of Simultaneous Pascal permit runtime preassignment of join locations for synchronization, preserving flexibility of application while minimizing synchronization overhead.

Rendezvous counters. Simultaneous Pascal synchronization requires that all threads within a particular parallel block complete before execution continues beyond the block. The order of completion is unimportant; so is the knowledge of which threads have not yet completed.

In a Fork statement, the number of threads to be joined is known at compile time. The number of threads to be joined in a Forall statement can be determined at runtime by evaluating the upper and lower bounds of the Forall range. In both cases, the number of joining threads is known before any of the threads is created.

Whenever threads are created, a small piece of memory is allocated to track the joining of those threads. This piece of memory is the join location for the threads. The join location contains a counter which is initialized to the number of threads to be joined. The address of the join location is passed to each thread as it is created. As a thread completes execution, it atomically decrements and reads the counter. If the value is greater than zero, the join is not yet complete. However, when the counter reaches zero, the join is complete, and program flow proceeds out of the current block.

This mechanism for synchronizing the completion of a set of threads is inexpensive, even when performed in software. It provides the functionality needed to resolve the join problem by exploiting the semantics of Simultaneous Pascal.

Dynamic structures of the RCM. One remaining issue is how the join address is transmitted to the threads it will synchronize. Consider the following simple example:

```
forall i := 1 to 10 do
  a[i] := b[i] * c[i];
```

Each instance of the Forall body is a single thread, and the address of the join location is passed to each thread when the thread is created. A more complicated example may pose a problem:

```
forall i := 1 to 10 do
  forall j := 1 to 10 do
    a[i,j] := b[i,j] * c[i,j];
```

In this case, the address of the join location representing the outer Forall is passed to each instance of its body. However, these threads immediately create new threads, to which is passed the address of the join location representing the inner Forall. Upon completion of the various instances of the Assignment statement, how will the join for the outermost join point occur?

SPOC employs a tree structure that grows and shrinks in response to the nesting of parallel blocks. Every join location also contains a pointer to the join location of the thread that created it. These pointers represent a tree, with each join location pointing to its parent. Each thread in the system has a pointer to some join point which is a leaf node of the tree. As parallel blocks are entered, the tree grows with the addition of new join locations. As these blocks join, the locations are removed and the tree shrinks.

When a join occurs, a thread is created to continue execution beyond the parallel block. This thread is given a pointer to the parent of the join point just synchronized. This new thread will, in turn, either extend the tree by entering a new parallel block or reduce the tree by participating in another join operation. Figure 3a shows the join location tree prior to the execution of a parallel thread. Figure 3b shows the tree with the additional join location for the executing threads. When they complete execution, the join location will be deallocated, and the tree will again look like Figure 3a.

The tree of join locations reflects the dynamics of program execution. Each thread need only retain a pointer to one join point in the tree, without regard to the depth of parallel nesting. The pointer to the relevant join location is passed to a thread when it is created. Relevant join information is stored within the structure; the expansion and contraction of the tree reveals this information at the correct times.

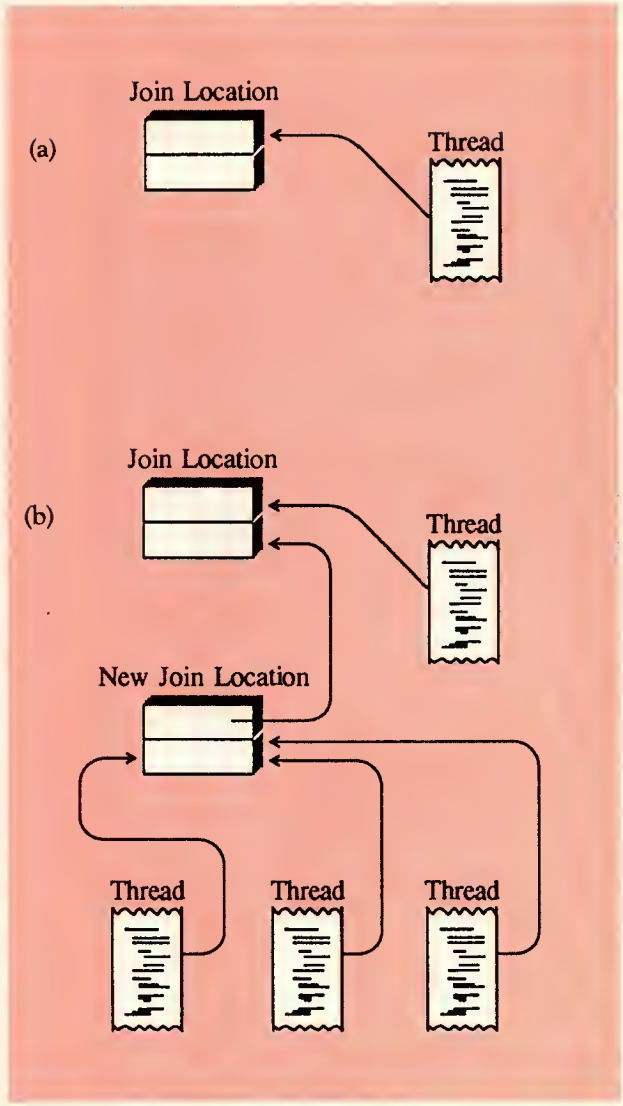


Figure 3. Join location before (a) and after (b) the Forall statement.

Memory management. The runtime needs of a Simultaneous Pascal program present a unique set of memory management problems for the runtime system. A program will require the standard Pascal heap, which is manipulated by the programmer using the New and Dispose intrinsic routines. In addition, a local stack mechanism is required so that runtime system routines can be accessed with minimal bus contention. Finally, each thread, procedure, or function requires a small piece of memory in which to retain any information local to that thread or routine. In order to meet these needs, memory is divided into three parts: the *heap*, the *local stacks*, and the *frame pools*.

Application program requests for dynamic memory allocation are satisfied from the heap. Any of several well-known algorithms for dynamic memory management can be used to implement the heap. All heap memory is located in the global memory space, so that pointers can be passed between processors.

Several libraries of routines in the runtime system handle I/O, arithmetic functions, system control, and the like. It is important that these routines run as fast as possible. Each processor in the system has a small stack located in the memory local to the processor. In order to reduce global-bus contention, parameters to these routines are pushed onto this stack whenever a library routine is called. In addition, these local stacks provide a place where processor interrupts can be fielded as quickly as possible.

Whenever a thread is created or a routine is called, a small piece of memory, called a *frame*, is associated with the thread or routine. For each thread, the frame contains back pointers to parent threads, the value of the thread index (if it was created by a Forall statement), and any local variables allocated within the thread by a Using statement. In the case of a routine, the frame contains both the static and dynamic back links for the routine, the return address parameters, and local variables. If a function is being called, space is also provided for a return value. Figure 4 shows the layout of the frames.

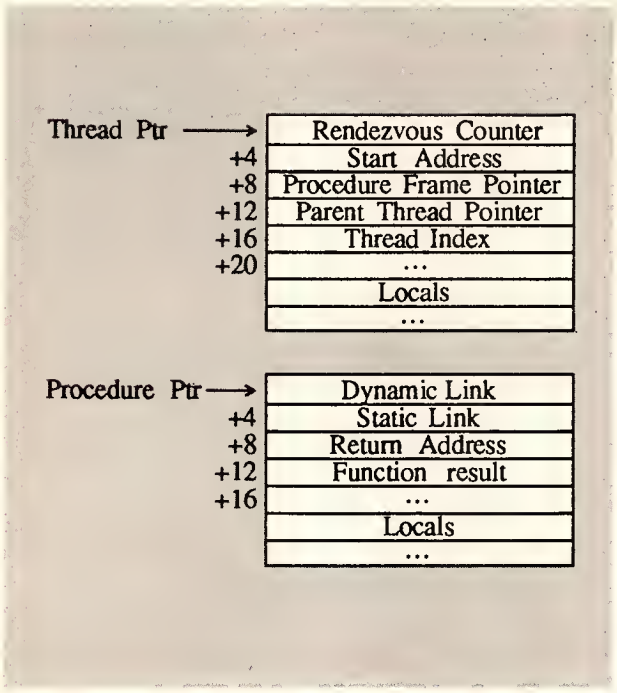


Figure 4. Layout of thread and procedure frames.

Several constraints are placed upon the frame pool. Many frames must be available, or the number of threads in the system at any moment will be reduced and the system will thrash for frames. The acquisition and release of a frame should occur quickly, since the speed of procedure calls and thread creation will be directly related to how quickly a frame can be made available.

The compiler and the runtime system jointly solve these problems. During compilation, the compiler determines the smallest usable frame size and passes this information to the runtime system. When the runtime system is initialized, a large chunk of memory is broken up into as many frames as possible, based on the size determined by the compiler. When a frame is required, it is removed from a list of free frames. When a frame is returned, it is placed back onto the free list. In order to reduce contention, the runtime system may allocate several disjoint frame pools, so that multiple processors can obtain frames simultaneously without contending for the same frame free list.

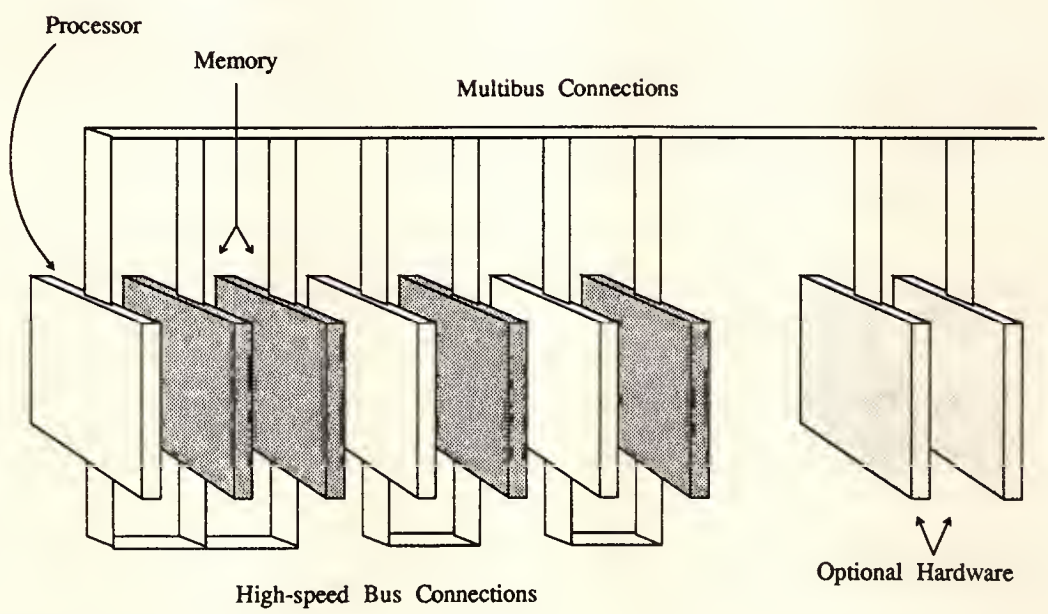


Figure 5. A Concert cluster.

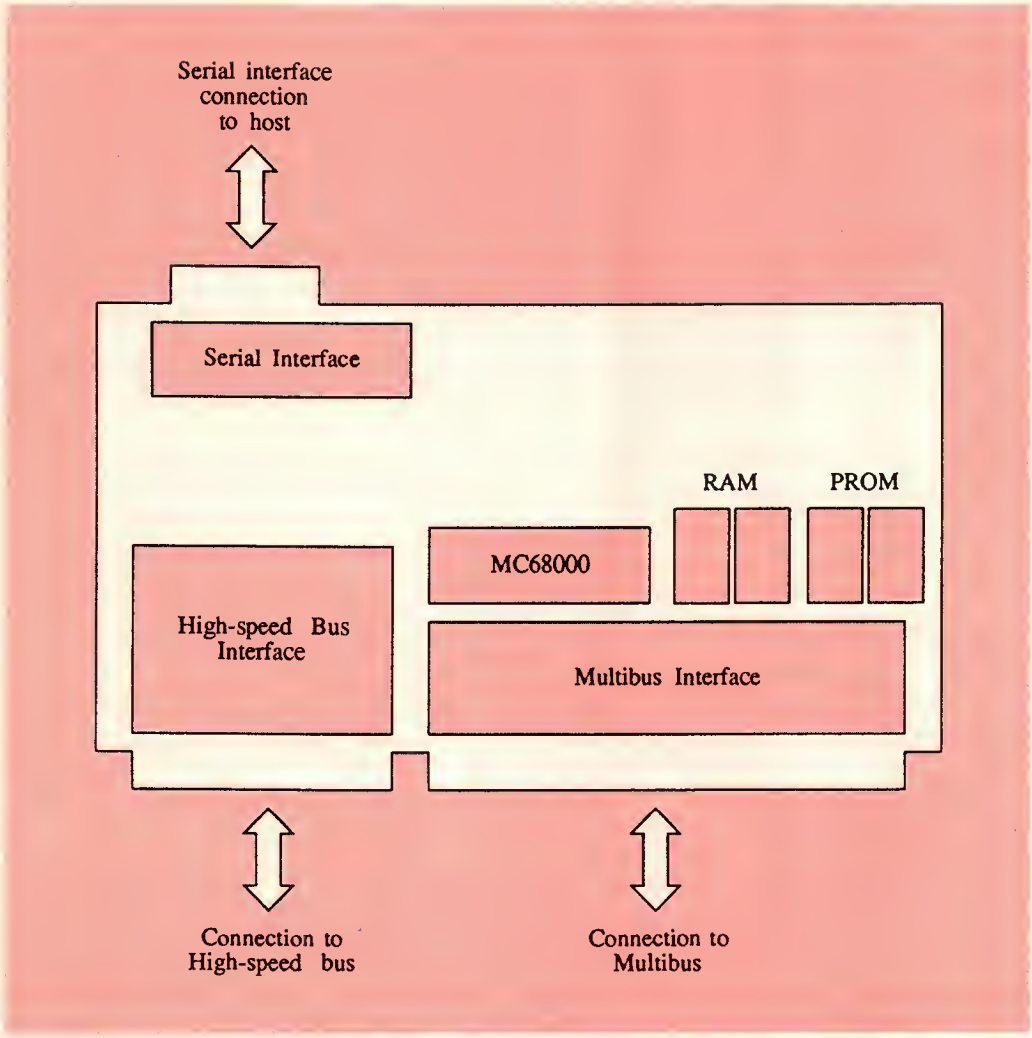


Figure 6. A processor module.

The Concert system

The Harris Concert multiprocessor is a tightly coupled, general-purpose computing resource that can support up to 64 processors. It differs from MIT's Concert machine[15] by using a crossbar switch rather than a ring bus interconnection scheme. Shared memory is intended to be the primary means of communication between processors. The Concert system hardware is shown in Figure 1.

The Concert structure exploits locality: processors and memory are grouped to decrease the cost of local accesses while (perhaps) increasing the cost of accesses to nonlocal resources. Groups of up to eight processors share a common backplane. Each backplane also supports several dual-ported memory modules, a global memory interface, and optionally a hard-disk controller, Ethernet controller, and/or custom system-monitoring hardware. This assembly is called a cluster (see Figure 5).

The cluster employs three levels of locality. Within a processor module (see Figure 6) the Motorola 68000 microprocessor has private access to 8K bytes of static RAM and/or PROM. The processor also has access via a private, high-speed bus to dual-ported RAM modules which share that bus. High-speed bus latency can be affected by Multibus accesses via the RAM module's other port, or by dynamic RAM refresh operations. The processor has access, using the Multibus, to the dual-ported RAM modules local to other processors in the cluster. The cost of this access can be increased by contention for the Multibus, by contention for the RAM module with its local processor, and by refresh cycles. Table 1 demonstrates various access costs and possible sources of contention.

A fourth level of memory hierarchy allows intercluster communication by providing a path from each cluster's Multibus, through a crossbar switch, to 8M bytes of interleaved global RAM (see Figure 7). The cost of accessing the global RAM can be affected by

Table 1.
Memory-access differences.

		Sources of contention			
RAM	Latency (ns)	Refresh	In same cluster	Other processor Using Multibus	In other cluster
High-speed bus	625	■	■		
On-board	750				
Multibus	1250	■	■	■	
Global	1500	■		■	■

Multibus contention, access to the same interleaved memory module by multiple clusters, and global RAM refresh cycles.

Results

The SPOC project will span several phases. The first phase, intended to prove the functionality of SPOC, is complete. We can currently create, compile, and execute Simultaneous Pascal programs on clusters of up to eight processors. Although our principal goal in the first phase was to achieve a functioning parallel processor, we also realized some performance gains. Programs which execute on several processors do exhibit performance enhancement as the number of processors increase.

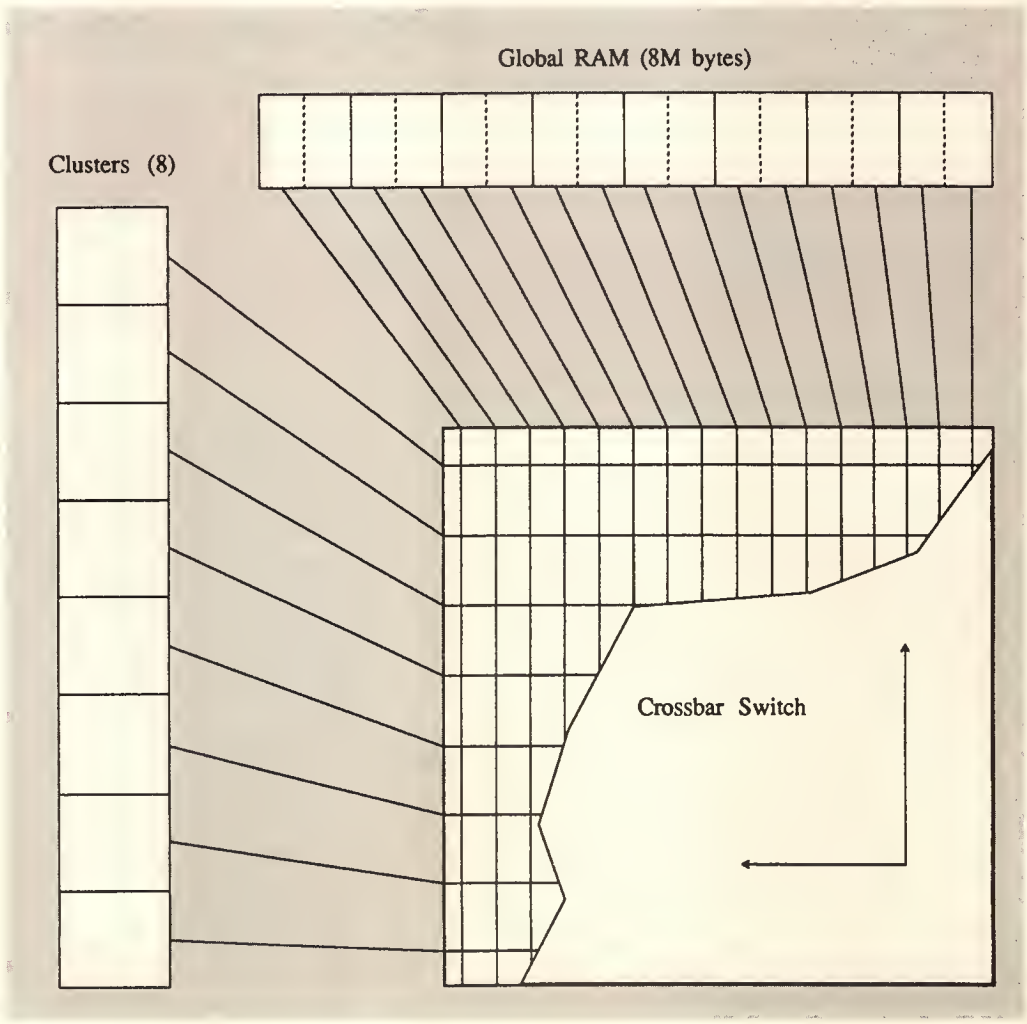


Figure 7. The Concert system.

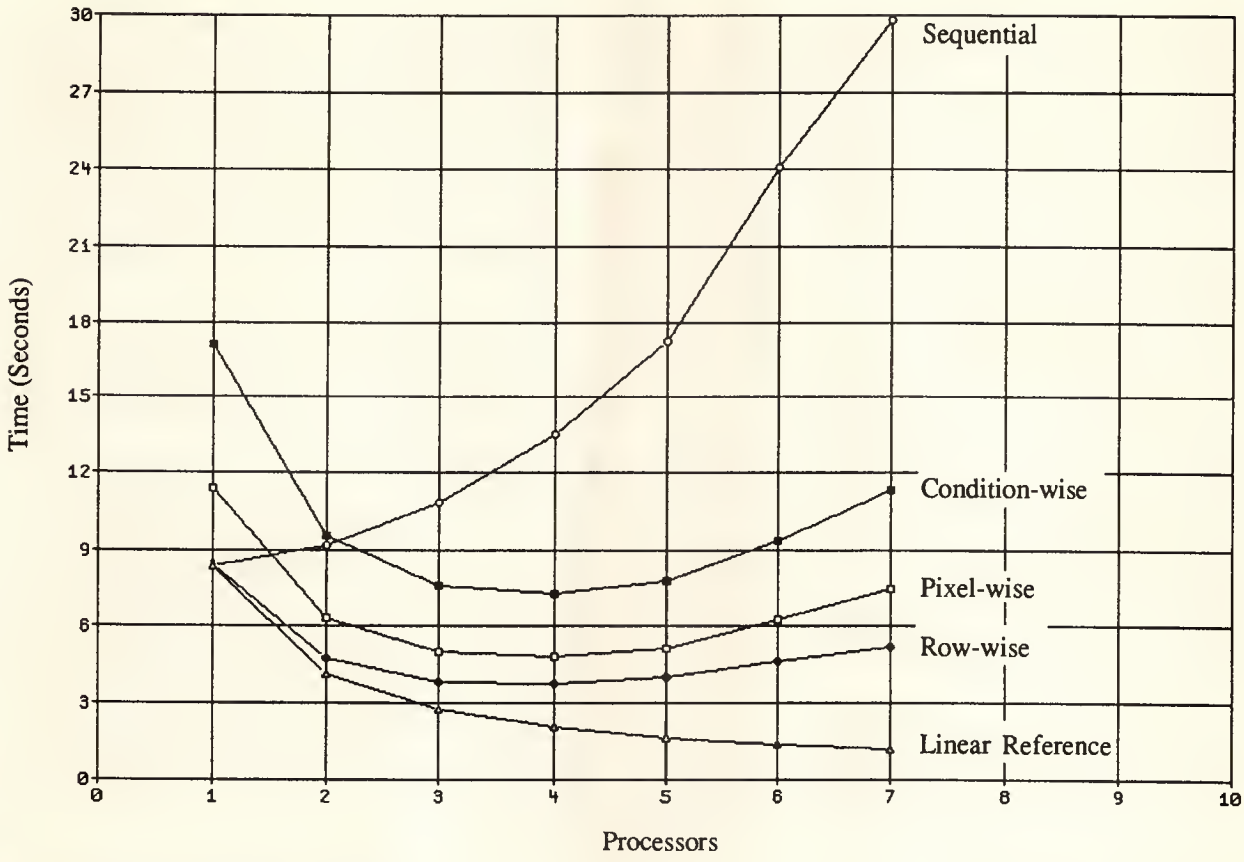


Figure 8. Digital-thinning performance.

We selected two applications to illustrate the effectiveness of the SPOC environment. The first is a digital-thinning algorithm,[16,17] used in optical applications to reduce complex images to their minimal, skeletal form. The second is Gaussian elimination, which is used in a variety of applications to solve linear equations.

Digital thinning. The digital-thinning algorithm is massively parallel and involves inspecting the neighbors of each pixel in an image. If the neighbors meet four different conditions, the pixel under consideration is allowed to remain in the image. Multiple passes are made over the image until it stabilizes and no more pixels can be removed.

The parallelism of this application is readily apparent. In all cases, each pixel of the image, and even each condition applied to each pixel, can be examined in parallel. Thus, the core of the algorithm can be described by the following fragment of code:

```
forall i := 1 to rows do
  forall j := 1 to columns do
    if image[i, j] then
      temp[i, j] :=
                neighbors(i, j)    |and|
                pattern(i, j)      |and|
                condition_c(i, j)  |and|
                condition_d(i, j);
image := temp;
```

Note the use of nested Forall statements to access each pixel of the image in parallel, as well as the parallel operator |and| to cause each of the conditions to be evaluated in parallel.

The performance graph of this algorithm in the SPOC system, relating execution time to the number of processors, is shown in Figure 8. The figure has several lines representing various thread granularities, a reference for the sequential version of the algorithm, and a reference representing the linear speedup expected of a perfect machine. A computing profile, which indicates the number of active threads at any given point, is shown in Figure 9. This profile indicates the parallelism available at each point during program execution.

The graph in Figure 8 gives tremendous insight into the performance of the SPOC system. Most interesting is the (rapidly!) climbing curve representing the sequential version of the program. When the sequential algorithm executes, one processor is performing

useful work while the remaining processors wait in vain for more threads to be created. These extra processors present a tremendous amount of global-bus contention, which hinders the progress of the sole working processor. This contention is particularly severe because the image being worked upon is held in global memory, along with the thread queue. Each atomic access to the thread queue by an idle processor precludes the use of the bus by the working processor. As more processors are added to the system, this contention increases.

Also of interest is the comparison between the versions of the program with different granularity. Because of its inherent parallelism, the digital-thinning algorithm is easily modified to be parallel in a row-, pixel-, or conditionwise manner. Currently, the amount of work performed for each condition, and thus for each pixel, is relatively small. Although each version of the algorithm shows some initial performance improvement when additional processors are added, the best scaling is achieved by the row-wise version, since it has the highest ratio of work performed versus scheduling cost.

Eventually, the addition of extra processors will begin to slow down the machine, as more time is spent waiting for and scheduling threads than is spent performing useful work. In addition, the extra bus contention created by extra processors, even though they are working, eventually bogs down the system.

Note that the image being thinned is only 52 × 48 pixels. Normal images in the optical realm would be orders of magnitude larger and would present significant amounts of work on a row-wise basis, providing for better scalability. Memory constraints within the current SPOC system prevented the use of a larger image.

Gaussian elimination. Gaussian elimination is a well-known technique for solving n linear equations in n unknowns. The algorithm consists of n - 1 phases, in which separate subphases pivot the matrix to bring the largest coefficient to the upper left, divide each element of the matrix by this coefficient, and finally subtract the upper row from each of the other rows. The pivoting phase is sequential in nature, while the division and subtraction phases can exploit either row- or element-wise parallelism. The code for each of these phases and subphases is shown in Figure 10. When all of these phases have been completed, a final series of parallel passes sets the upper triangle of the matrix to zero. The algorithm thus causes alternating periods of parallel and sequential activity.

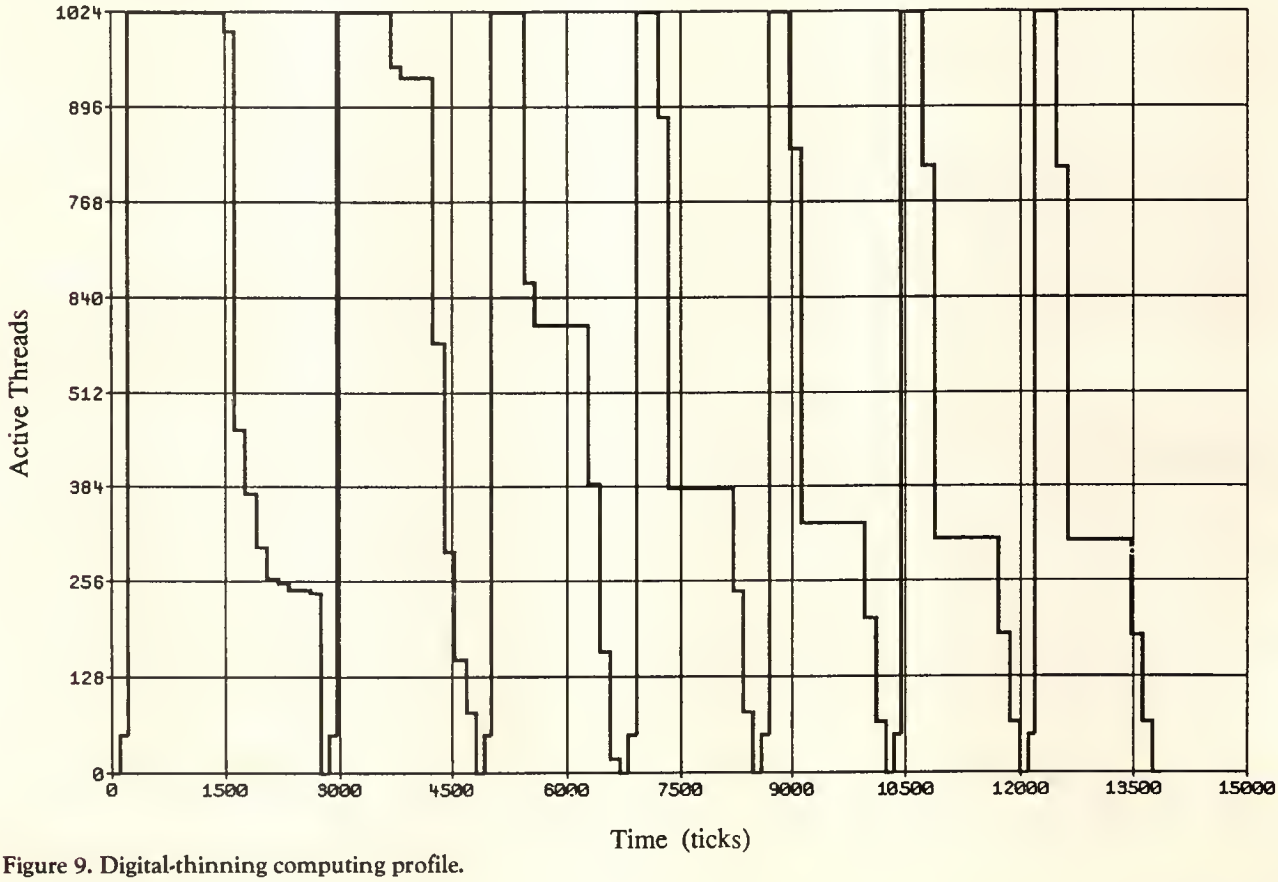


Figure 9. Digital-thinning computing profile.

```
for submat := 1 to size - 1 do
  begin
  { pivot the matrix }
  for i := submat + 1 to size do
    if abs(a[submat, column[i]]) > abs(a[submat, column[submat]]) then
      begin
      temp := column[submat];
      column[submat] := column[i];
      column[i] := temp
      end;
  { divide each row by the value at the upper left corner }
  forall i := submat to size do
    if (a[i, column[submat]]<>0.0) and (a[i, column[submat]]<>1.0) then
      forall j := submat + 1 to size + 1 do
        a[i, column[j]] := a[i, column[j]] / a[i, column[submat]];
  { subtract the top row from each of the others }
  forall i := submat + 1 to size do
    if a[i, column[submat]] <> 0.0 then
      forall j := submat to size + 1 do
        a[i, column[j]] := a[i, column[j]] - a[submat, column[j]];
  end
```

Figure 10. Simultaneous Pascal version of Gaussian elimination.

We tested two versions of the program, one exploiting element-wise parallelism, the other using row-wise parallelism. The performance graph and computing profile are shown in Figures 11 and 12.

Again, the sequential version suffers from bus contention in a manner similar to the digital-thinning algorithm. The losses are not so dramatic, however, because the Gaussian elimination program spends most of its time performing floating-point arithmetic. The library routines execute only in local memory and are not impacted by global-bus contention. Contention does occur, though, when a processor is fetching operands to pass in to the library routines.

The scalability of the Gaussian elimination program is much better than the digital-thinning program, because the smallest thread has more work to perform. Thus, the ratio of useful work to scheduling costs rises, and the scalability improves.

As in digital thinning, the row-wise version of the program scaled best and was still improving at seven processors. It most likely would have bottomed out around eight or nine processors.

Observations from experience. Although the previous examples show some weakness in the current implementation of SPOC, we obtained encouraging results even before we made performance modifications to the system. The fact that the system shows near-linear speedup for a small number of processors indicates that the goal of moderate scalability is obtainable.

Significant performance losses are occurring in at least two areas within SPOC. The first is global-bus contention caused by multiple processors attempting to gain exclusive access to the various thread and frame queues in the system. The performance curves of the two example sequential versions in Figures 8 and 11 demonstrate the effects of the unoptimized, runtime system software. The second source of performance loss is the overhead cost of scheduling a thread. Different performance curves representing various levels of granularity illustrate the impact of this overhead in Figures 8 and 11. The coarsely grained versions of each application ran more quickly than their finely grained counterparts.

The primary goal of the next phase of SPOC is to enhance the system's performance characteristics. This includes finding new mechanisms to prevent the bus and queue contention now occurring in the system, as well as reducing the time required to create and schedule a thread. Although our current efforts are directed toward software solutions to these problems, eventually hardware support for these operations may provide the best answer.

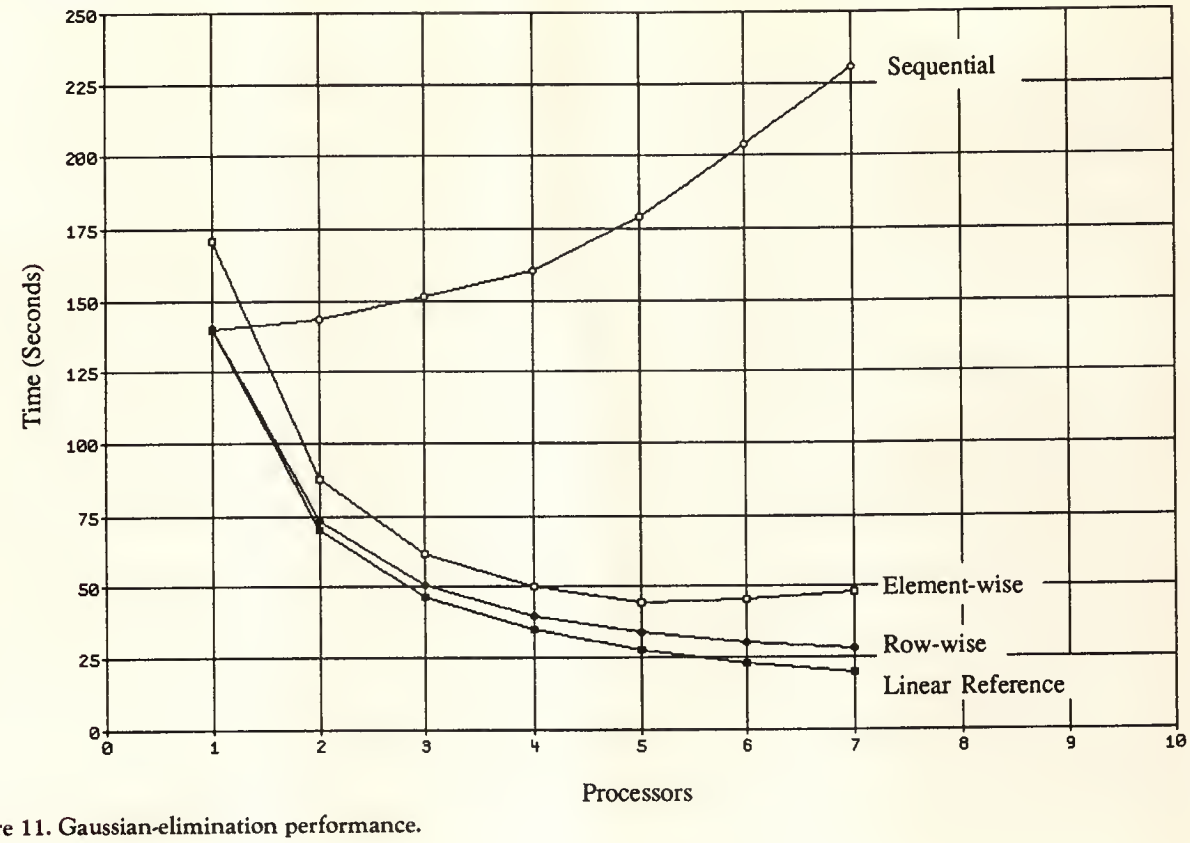


Figure 11. Gaussian-elimination performance.

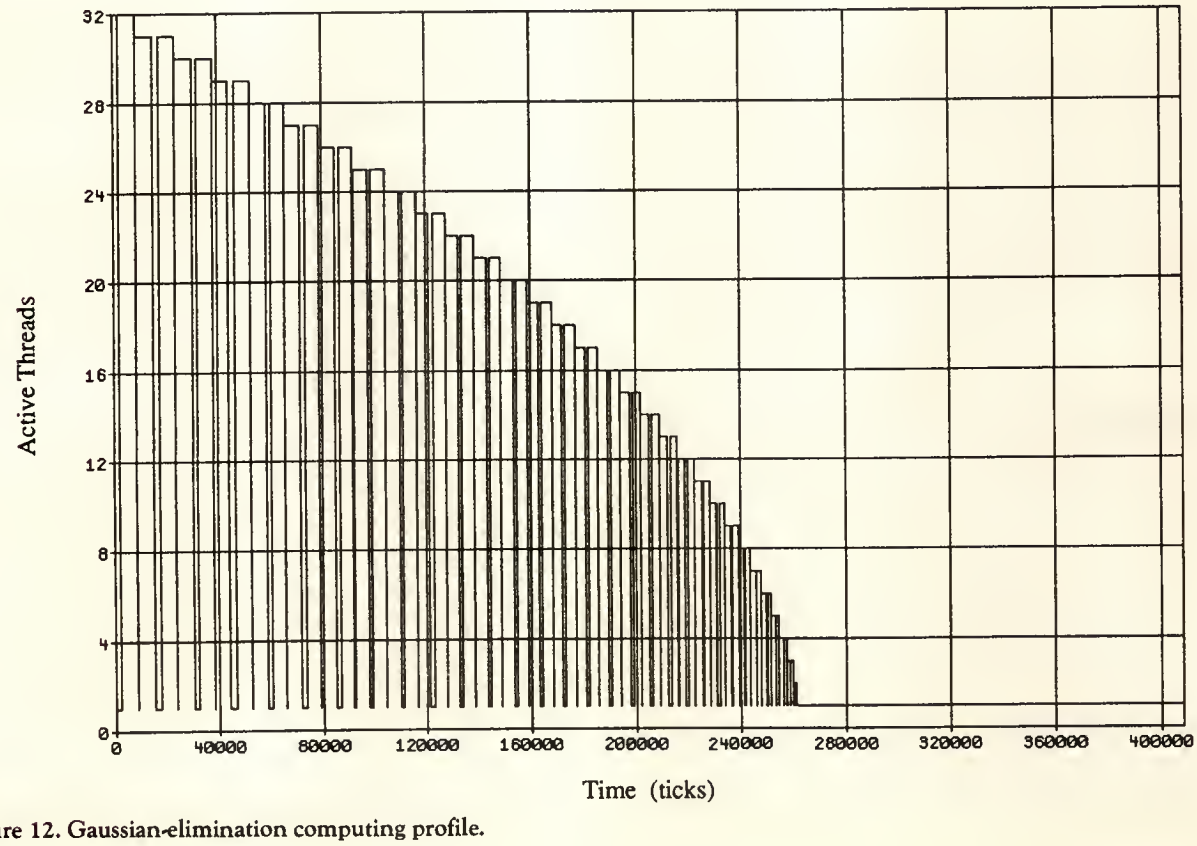


Figure 12. Gaussian-elimination computing profile.

The SPOC user environment

The ultimate value of a multiprocessor is dependent not only on how fast it executes a program but also on how easily the application programs can be created, debugged, and run. The SPOC multiprocessor environment includes an array of tools to facilitate rapid program development and support convenient user interaction. This environment encompasses both the Concert multiprocessor and its DEC VAX 11/750 host with Unix Berkeley 4.2, providing a rich set of tools for program development. It includes the host's Simultaneous Pascal software development environment, Concert's Simultaneous Pascal program execution environment, and the SPOC user interface.

The host Simultaneous Pascal environment. The host machine provides several tools to assist in the development of parallel programs. These tools include a compiler to generate serial versions of parallel programs suitable for host emulation, along with a graphics generator to produce computing profiles for the emulated code. Other programs are available in the host environment to detect parallel semantic errors, such as race conditions when accessing shared variables. Finally, the host machine contains the Simultaneous Pascal cross-compiler to generate parallel code suitable for execution on the Concert multiprocessor.

The Concert Simultaneous Pascal environment. The Simultaneous Pascal program execution environment on the Concert multiprocessor consists of four major subsystems: the SPOC user interface, the Concert runtime system, the debugger, and hardware instrumentation for performance analysis.

The SPOC user interface. The SPOC user interface (Figure 13) provides users with their primary impressions of the system. It performs three functions:

- It removes the burden of system housekeeping and initialization from the experimenter, who may be unfamiliar with the system.
- It performs file transfers and session recording at the user's request.
- It supports debugging tools which are invoked at the user's request or when a fault occurs on a processor in the SPOC configuration.

The available debugging tools include many low-level functions needed by the SPOC system implementers, who use the same user interface while doing SPOC development work. As the system matures, we expect the implementers to become experimenters themselves. The unified environment provided by the tools will allow the experimenters to move easily between implementing new features and evaluating changes in performance. The implementers' experience and the immediate feedback to their own work may well result in some of the most direct and valuable parallel-processing insight SPOC will produce.

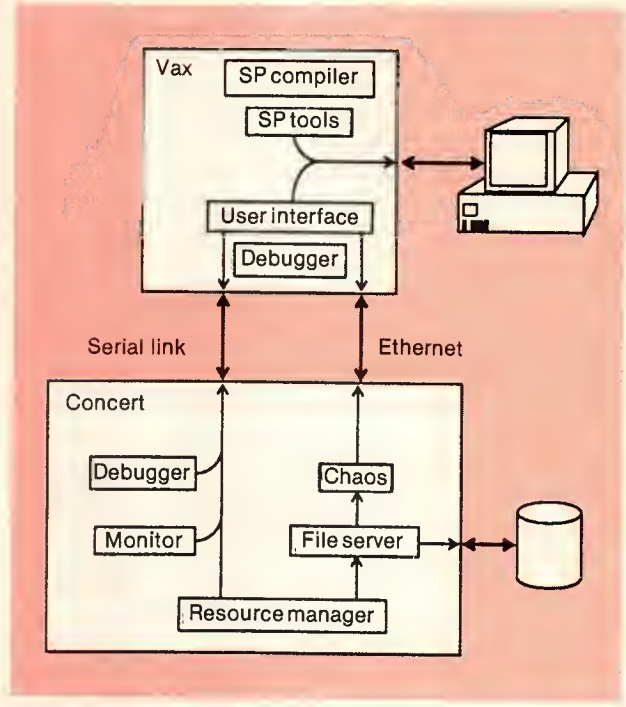


Figure 13. The SPOC user interface.

Limits of Simultaneous Pascal semantics

Simultaneous Pascal is an effective tool for representing many algorithms in a way that captures much of their available parallelism. The language and the underlying RCM are proving to be practical except for the following deficiencies.

List operations. The Forall construct permits the same action to be performed in parallel on all elements of arrays and sets. Currently, however, a Forall-like operation cannot be performed on the elements of a linked list. Because list processing is important for symbolic computation, the inability to apply a parallel operator to such data structures is an unfortunate limitation. Difficulty arises from the dynamic character of lists: Their length can only be determined by tracing pointers.

It would be desirable to augment Simultaneous Pascal with a construct that allowed parallel operations over lists and other dynamic data structures. Such a construct would allow the programmer to traverse a data structure, creating a thread for each element of the structure. A future version of Simultaneous Pascal may include this feature.

Overly constrained structure. Simultaneous Pascal encourages a programming style that promotes *chunky* parallelism resulting in hourglass-like computing profiles. Precedence constraints for a thread are often satisfied when only a few of the threads in a preceding parallel statement have completed. Since all the threads in a parallel block must terminate before any of the succeeding threads may begin, the execution of subsequent threads is unnecessarily delayed. This behavior can restrict the parallelism available in an algorithm. The impact of these constraints on execution behavior merits further study.

Artificial data dependencies. Program flow in Simultaneous Pascal is tied directly to the control relationships expressed in the program. Although this is satisfactory for many applications, the control-flow model can be artificially constraining. The Future construct in MultiLisp[18] provides a means for removing unnecessary program-flow constraints, leaving only the critical data dependencies. If a primitive operation needs an operand value before it has been computed, the thread is suspended. Once the value has been determined, the old thread is reawakened.

This form of parallelism is appropriate in symbolic applications where large, compound-data structures are manipulated in an unpredictable manner. Data-manipulation primitives that do not inspect the value will not be blocked on an uncomputed value. Programming with partially computed values, as in building data structures in which only some of the values have been calculated, becomes possible. This data-driven synchronization provides many opportunities for overlapped computation that cannot be captured in Simultaneous Pascal.

Granularity of parallelism. While too little algorithmic parallelism results in poor efficiency due to low processor utilization, too much parallelism can be equally detrimental to multiprocessor performance. Having too few processors results in the serialization of potentially parallel threads, while retaining the overhead associated with those threads. This serialization can substantially reduce the effective performance of a medium-scale system. A technique called *aggregation* would allow the compiler to build fewer, more coarsely grained threads from many finely grained ones. Aggregation would eliminate excessive overhead losses while maintaining sufficient granularity to retain good load-balancing.

A second problem is a parallelism explosion which reveals so much parallelism at one time that the system's capacity to keep track of all the pending threads is saturated. Representing large groups of similar threads with a single object in the system can minimize this occurrence. As processors request threads to execute, small numbers of threads can be created, with their (as yet) uninstantiated siblings represented by the object.

Unfair thread assignment. An objective of Simultaneous Pascal and SPOC is to eliminate the overhead costs of context switching incurred by suspending tasks. This approach is in sharp contrast to some CSP-based multiprocessors, the use of I-structures in dynamic data flow, and the use of Future in MultiLisp. A characteristic of Simultaneous Pascal is to assume that a thread, once scheduled for execution on a specified processor, will reside on that processor until it has completed. Processors are not time-sliced among active threads. This characteristic constrains the style of programming and eliminates the use of mutual exclusion as a means of interthread synchronization. All precedence constraints of a thread must be satisfied prior to the thread being scheduled for execution. A negative consequence of this unfair scheduling policy is that deadlocks can occur if CSP-style synchronization of communication between two or more threads is attempted.

Current state of SPOC development. We have achieved our goal of assembling a system that will accept Simultaneous Pascal source code, compile it to run on a sequential or a parallel machine, cause it to be executed on the desired machine, and produce statistics indicating the effectiveness of the program. We plan to continue enhancing performance. The current system is intended as a functional proof of concept and is not a high-performance vehicle. As SPOC performance improves, the techniques that evolve will determine new approaches to the design of parallel system hardware.

This last result is of particular importance to the parallel processor architect. Multiprocessors are implemented with off-the-shelf microprocessors because they are readily available. These processors are not designed to be constituent elements of large multiprocessors, and as a result provide little support for parallel processing. SPOC can help determine the nature of that support and the potential advantage of having that capability available in hardware instead of in software. The next generation of multiprocessors will include special hardware to efficiently control parallel execution.

Efficient control of parallel activities remains one of the critical unresolved problems inhibiting the application of multiprocessors to general-purpose computing. The SPOC multiprocessor execution environment embodies an experimental approach that integrates semantics of parallel control with mechanisms for synchronization of concurrent tasks on a medium-scale multi-microprocessor.

SPOC executes programs written in the experimental parallel programming language, Simultaneous Pascal. The parallel control constructs in Simultaneous Pascal are the Forall and Fork statements. Also, parallel operators permit subexpressions to be evaluated concurrently. The Locking statement enables exclusive access to shared data objects. The Using statement allows fine-grained scoping of temporary variables without resorting to procedure calls.

SPOC synchronizes the termination of concurrent threads of a parallel statement with the RCM. When a parallel statement is executed, a counter is allocated and used to keep track of the number of terminating threads, providing a means for synchronization. Since parallel statements can be nested, all the counters are linked in a tree structure that reflects the dynamic state of statement nesting. The cost of managing the counters is low, even when implemented in software. The RCM fully supports the synchronization requirements dictated by the parallel-control semantics of Simultaneous Pascal.

SPOC programs execute on the Concert multiprocessor. It incorporates 64 16-bit, MC68000 microprocessors, each with 512K bytes of local memory, and 8M bytes of shared memory. The machine is divided into eight clusters, each containing eight processors. Each cluster has a high-speed common bus and is connected to the global memory by means of a crossbar switch. Additional global registers provide for system-wide interrupts. SPOC is integrated with a host VAX 11/750 via Ethernet. Cross-compiler, debugger, and user-interface tools reside on the host and provide easy access to SPOC facilities.

The SPOC project is a relatively new, yet fully functional, system. Current activities include optimizing the runtime system code to minimize losses due to shared resource contention and extending the base of application programs used to test SPOC. Preliminary results show acceptable performance gains for small numbers of processors. We anticipate that future enhancements to SPOC will improve scalability, allowing full use of Concert multiprocessor resources.

Acknowledgments

We wish to recognize the contributions made by our colleague Michael D. Noakes in the formulation of the initial SPOC concept and in the implementation the SPOC resource manager. We also wish to acknowledge the leadership and guidance provided by Robert H. Halstead of MIT during the initial development of Simultaneous Pascal and the Concert multiprocessor. Finally, we wish to convey our thanks to our colleagues in the Advanced Technology Department at Harris Corporation for their many valuable suggestions during the creation of SPOC.

References

1. M.J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers*, Sept. 1972, pp. 948-960.
2. R.J. Swan, S.H. Fuller, and D.P. Siewiorek, "Cm*—A Modular, Multi-Microprocessor," *AFIPS 46, Proc. 1977 NCC*, AFIPS Press, Montvale, N.J., pp. 637-644.
3. T.L. Sterling, "Parallel Control Flow Mechanisms for Dynamic Scheduling of Tightly Coupled Multiprocessors," PhD thesis, EECS Dept., MIT, Cambridge, Mass., May 1984, pp. 24-29.
4. T.L. Anderson, "The Design of a Multiprocessor Development System," Tech. Report MIT/LCS/TR-279, Lab. Computer Science, MIT, Sept. 1982.
5. W.A. Wulf and G.C. Bell, "C.mmp – A Multi-mini-processor," *Proc. AFIPS 1972 Fall Jt. Computer Conf.*, AFIPS Press, 1972, pp. 765-777.
6. N. Matelan, "The FLEX/32 Multicomputer," *12th Ann. Int'l Symp. Computer Architecture*, CS-IEEE, Los Alamitos, Calif., June 1985, pp. 209-213.
7. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM 21*, Prentice-Hall Int'l, London, England, Aug. 1978, pp. 666-677.
8. N. Gehani, *Ada Concurrent Programming*, Prentice-Hall, Inc., New York, N.Y., 1984.
9. Inmos Ltd., *Occam Programming Manual*, Prentice-Hall Int'l, 1984.
10. P. B. Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., 1977.
11. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor," *Proc. Second Ann. Symp. Computer Architecture*, CS-IEEE, Los Alamitos, Calif., Dec. 1974, pp. 126-132.

12. W.B. Ackerman, "Data Flow Languages," *Computer*, Feb. 1982, pp. 15-36.

13. J. Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm. ACM*, ACM, New York, N.Y., Aug. 1978, pp. 613-641.

14. D. Cooper, *Standard Pascal User Reference Manual*, W.W. Norton & Company, New York, N.Y., 1983.

15. R.H. Halstead et al., "Concert: Design of a Multiprocessor Development System," *13th Ann. Symp. Computer Architectures*, CS-IEEE, Los Alamitos, Calif., June 1986, pp. 40-48.

16. T.Y. Zhang and C.Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns," *Comm. ACM 27*, ACM, Mar. 1984, pp. 236-239.

17. H.E. Lu and P.S.P. Wang, "A Comment on 'A Fast Parallel Algorithm for Thinning Digital Patterns,'" *Comm. ACM* 29, ACM, Mar. 1986, pp. 239-242.

18. R.H. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Trans. Programming Languages and Systems*, ACM, Oct. 1985, pp. 501-538.

Thomas L. Sterling is a senior principal engineer in the Advanced Technology Department of Harris Corporation's Government Systems Sector. His interests include parallel-control flow models of computation, high-level language parallelism constructs, static data-flow architectures, and computational dynamics.

Sterling received his PhD in electrical engineering from the Massachusetts Institute of Technology in 1984.

Ellery Y. Chan is a lead engineer in the Advanced Technology Department of Harris Corporation's Government Systems Sector. Prior to 1984, he was a hardware designer at Atex Incorporated. His current research interests include computer graphics, parallel processing, and user-interface design.

Chan received a BS in computer science and engineering from the Massachusetts Institute of Technology in 1981.

Albert J. Musciano is a lead software engineer in the Advanced Technology Department of Harris Corporation's Government Systems Sector and has been with Harris since 1982. His current research interests include parallel processing, language and compiler development, user-interface design, and software tools development.

Musciano received a BS in information and computer science from the Georgia Institute of Technology in 1982.

Douglas A. Thomae is a lead engineer in the Advanced Technology Department of Harris Corporation's Government Systems Sector and has been with Harris since 1982. His current research interests include parallel architectures and functional languages.

Thomae received a BS in electrical engineering from the University of Wisconsin-Madison in 1982.

Questions about this article can be directed to Musciano at the Harris Corporation, PO Box 37, MS 3A/1912, Melbourne, FL 32902, or via electronic mail to chuck@harris-trantor.arpa.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 159 Medium 160 High 161

An Application-Specific Coprocessor for High-Speed Cellular Logic Operations

Robert E. Jenkins and D. Gilbert Lee, Jr.
Johns Hopkins University Applied Physics Laboratory

Why develop a special coprocessor to perform one lengthy computation? Performance is much better than with equivalent software and fast-turnaround development times will soon be possible.

The maturing technology in full-custom VLSI chips significantly affects the alternative approaches available for processing tasks. Application-specific chips, designed and fabricated inexpensively and quickly, often provide a viable alternative to software implemented on a general-purpose mainframe or a microcomputer. The use of custom VLSI chips makes a complex processor small enough and its power consumption low enough to be comfortably hosted by a personal computer. The personal computer provides a nicely suited front end for interactive use of the processor. This combination is a natural extension of the old idea of an attached array processor for matrix computations or even a floating-point coprocessor chip for PC-sized systems. The difference is that with custom VLSI one can think in terms of a high-performance coprocessor designed for a quite narrowly defined computing problem.

Does it make sense to develop a special coprocessor to perform, say, a specific Kalman filter computation and nothing else, rather than write a software version for a general-purpose computer? It does, but only if (1) the chip can be produced in about the same order of magnitude of time and cost as needed to develop equivalent software, and (2) it has a marked improvement in performance over software.

Chips designed with silicon compilers and fabricated via a fast-turnaround foundry service are beginning to make the application-specific coprocessor a sensible idea for a computing task that does not have a wide commercial market. At Johns Hopkins we have been exploring this potential through example projects to develop some narrowly defined processors at the very lowest end of the cost spectrum and then assess the usefulness of the results. Ideal candidates for such an approach are well-defined computations that will be run a large number of times—and which are formidable enough to cause the computing time on a programmable machine to be troublesome.

One of our selected computational tasks for an application-specific coprocessor is the cellular logic operation, or CLO, which is a special case of cellular automaton processing. Cellular automata are dynamical systems (introduced in 1948 by J. von Neumann and S. Ulam) made up of a regular array of sites, or cells, which can each be in one of a finite number of states. The system evolves to new states in discrete time steps through the interaction of each cell with its nearest neighbors. Such sys-

0272-1732/87/1000-0063$01.00 © 1987 IEEE

tems have been studied in full generality. However, most practical applications thus far have dealt with a two-dimensional array or grid, with a small number of possible states for the cells. In most such practical systems the state of a cell at time $t+1$ depends only on its state and those of its eight neighbors at time t. t can be expressed as the function:

$$X_{i,j}^{t+1} = F\left(X_{i,j}^{t}, X_{i+1,j}^{t}, X_{i-1,j}^{t}, \ldots, X_{i,j-1}^{t}\right). \quad (1)$$

The notation

$$X_{i,j}^{t} \quad (2)$$

denotes the state of site (i,j) at time t.

If the number of possible states is two, we have the special case of a binary array, where the function F becomes a 9-bit Boolean mapping referred to as a cellular logic operation. One defines this mapping by completely specifying the table of 2^9 possible states in the function. This binary form of automata represents a black-white image and has been used in this form for extensive image processing applications.[1,2] This special case is also the form for the well-known Game of Life neighborhood rule, invented by J. Conway in 1970, in which the individual cells live and die as a result of the number of neighbors.[3]

Research in the application of cellular automata has been ongoing for some time and is more active than ever. Interesting and promising new approaches based on cellular automata models have been proposed in a wide variety of physical and biological modeling problems, ranging in nature from galactic structure[4] to DNA sequences.[5] A representative sample of the ongoing research may be found in Farmer et al.,[6] which includes a good overview by S. Wolfram in the preface. Applications to image analysis are very well developed.[1,2] Preston at Carnegie Mellon and his collaborators at Perkin-Elmer have produced a number of parallel image processors based on cellular logic operations, the latest called the PHP.[7] Chapter 10 of Preston and Duff offers a good survey of machines developed in the past two decades that can utilize the inherent parallelism of cellular logic operations on two-dimensional images.

As with image processing, researchers studying CLO applications have an important need to develop algorithms in the interactive mode, with graphical displays of the arrays. Toffoli[8] points out the general need for high-speed computing hardware to support the interactive exploration of cellular automata applications. Toffoli has developed a "Cellular Automaton Machine" for this purpose. In the spirit of this project, we have addressed this problem from a somewhat different approach than that taken for Toffoli's machine, Preston's PHP, or other recently developed processors.[7] Our goal is to achieve the least expensive CLO processor that still satisfies some minimum requirements of usefulness for a researcher.

By "least expensive," we mean we settled on a processor that would cost less than $500 and would be completely supported by a stand-alone IBM Personal Computer with a graphics board. We also required that the entire processor would fit on a single board in the PC. This cost goal easily brings the machine within the realm of any interested graduate students or independent researchers wishing to explore some cellular automaton ideas in the interactive graphics mode in their offices or homes. Such a computing task is not of wide commercial appeal; however, it is of great interest to a limited class of users.

We achieved high performance for very small cost by

- making the architecture narrowly focused and application specific,
- pipelining to obtain a reasonable level of parallelism, and
- using custom 3-micron chips for implementation.

Here we present the details of the processor along with its resulting performance.

Processor capability and architecture

A critical factor in realizing a practical application-specific processor is the early definition of a very specific form for the computation that is to be performed. The introduction of a lot of generality and flexibility into the design at extra cost obviously defeats the basic idea. We defined the computation for the CLO processor by

- selecting an arbitrary but reasonable array size,
- restricting the states to two, and
- applying the constraint of supporting interactive graphics displays. The processing speed needed for this constraint is difficult to achieve with software implemented on a general-purpose machine.

Conway's single-bit Game of Life provides a rich, standardized problem domain for cellular-automata research—and continues to stimulate a lot of ideas. Although processors for multibit cells had already been developed,[8] we felt that the two-state automata will continue to be useful for exploring new ideas for quite a while. We thus settled on a single-bit processor performing neighborhood logic operations on 256×256 arrays to greatly reduce the scope. The operation is based on the 3×3 neighborhood (that is, square tessellations). This neighborhood allows rules in the so-called von Neumann and Moore neighborhoods as well as hexagonal tessellation as a special case.

Enough processor speed and display speed to support interactive computing is much more important to the usefulness of a cellular automaton processor than the restriction to binary arrays. Toffoli's discus-

sion of this point suggests that something close to video rates (30 array transitions/second) is a useful display speed in this regard so that researchers can watch a full 2D array graphically "evolve under our eyes." This capability for a 256 × 256, single-bit array is just about satisfied by DMA transfers between the auxiliary cellular logic processor and the host PC memory. These transfers take about 50 milliseconds for the full array on the IBM PC.

For many applications, such as a sophisticated pattern-recognition algorithm, hundreds of CLO steps may be needed before a meaningful new state emerges. Consequently, we felt it was necessary to have the additional capability to compute transitions faster than video rate and then intervene periodically to update the graphics display. We achieved such a processor throughput in a simple architecture with a 12-transformation pipeline structure.

In this structure a cellular logic transformation on the rasterized array is performed at each sequential stage by a single, independently programmable, custom VLSI chip. Twelve such chips fit nicely on the processor board along with DMA servicing logic, and a 4-MHz clock is sufficient to allow the bit serial chip pipeline to keep up with the DMA byte transfers. The DMA bytes are serialized, and the bit stream snakes through the pipeline with the output of each (identical) chip being the input for the next. The host computer halts during the pipeline operation to maximize the DMA rate and is revived by an interrupt generated at the completion of the process.

This architecture allows 12 sequential state transitions of the complete 256 × 256 array in about 50 milliseconds, with the final state emerging from the pipeline via DMAs back to the computer memory or graphics memory. The parallelism introduced by pipelining 12 successive transformations helps keep the DMA transfers from becoming a computational bottleneck. The resulting effective processing rate of 240 array transformations per second can of course only be realized in a sustained mode for a problem in which the neighborhood rules for each of the 12 sequential transformations do not change from cycle to cycle. In cases requiring rule changes, the software intervention to load new transformation tables between processor calls adds a small additional delay.

The obvious key element in the processor is the custom chip that performs the cellular logic operations. This chip was developed as a class project in a VLSI course at Hopkins last year and was successfully fabricated through the MOSIS system at USC. MOSIS is a fast-turnaround chip fabrication service, initiated by the US Department of Defense Advanced Research Projects Agency and operated by the Information Sciences Institute at the University of Southern California. This service has greatly reduced the fabrication costs for small-volume projects by introducing process standardization and the multi-project wafer concept. In reasonable quantities the chips could be manufactured for less than $20 apiece.

CLO chip architecture

We designed our chip to perform, in a pipeline, a programmable square tessellation CLO transformation on a rasterized, single-bit array via a 64-byte lookup table to specify the Boolean mapping. The architecture is based on the original ideas presented by Golay[9] and their later implementations by the Perkin-Elmer group, although we have implemented the more-general square rather than the hexagonal neighborhood.

In addition, the chip keeps a count of the number of bits that are changed during the complete transformation. This count is an important piece of information for automating things such as image processing. For example, to perform skeletonization on an image that is an important preliminary step in pattern recognition, we apply the appropriate transformation repeatedly until the skeleton is stable. This stability is indicated by zero bits changing when the process is complete. The "bits-changed" counter for each chip is easily accessible as read-only memory locations mapped into the PC memory space.

The on-chip lookup table is a 512-bit static RAM accessible to the user as 64-byte read/write tables in the host PC memory space. During pipeline operation, as each bit is processed, its 3 × 3 neighborhood is used to create a 9-bit address into the lookup table. The high- and low-order bytes of the bits-changed counter are readable as table entries 65 and 66. Once a complete array has been processed, the bits-changed counter holds its value until it is reset by an external Sync signal that readies the chip for the start of a new array.

We used the standard approach of having the input array serially clocked into a 515-bit shift register (two complete rows plus three bits). The 9-bit lookup address is formed by judiciously selected cells along its length. The transformed bit is output directly into the next chip in the pipeline. A position (row/column) counter counts each input bit to keep track of array edges and the state of the pipeline so that various required control signals can be generated. One of these is a first-bit-out output signal that becomes the Sync (or start) signal for the next chip.

Two border-control chip inputs specify how to treat bits on the edges of the array. Four options are available: do nothing, invert the bit, make it a zero, or make it a one; the action is applied to the incoming bit stream. (If the do nothing option is selected, the architecture essentially wraps the array around the vertical axis to form a spiral of the horizontal rows.)

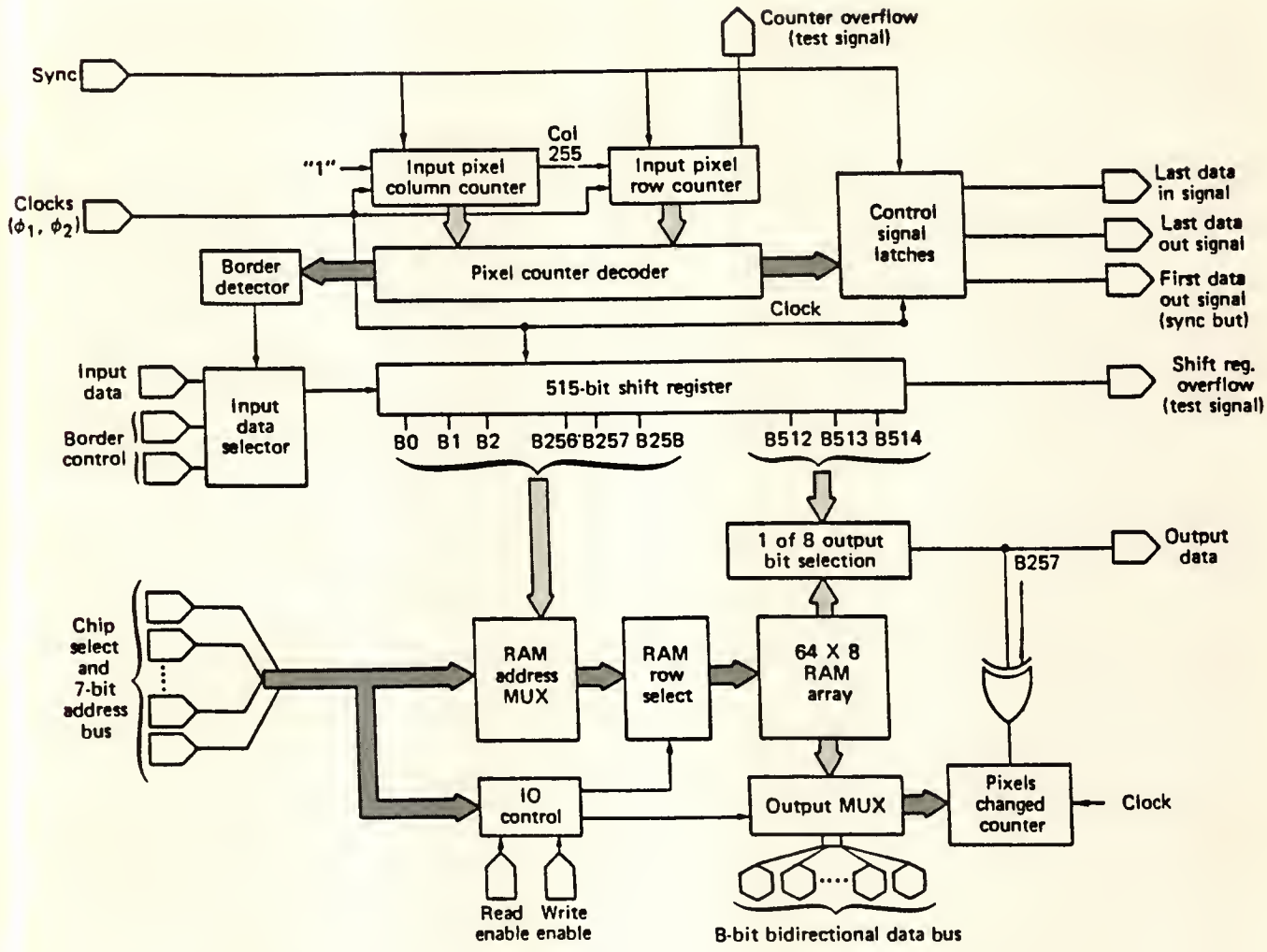


Figure 1. CLO chip architecture.

The chip architecture is shown in Figure 1. It is an NMOS design using one of the MOSIS standard 40-pin frames. The functional chips we received worked up to 10 MHz, which exceeded our requirements for an IBM PC coprocessor.

Architecture of processor board

The processor board contains: (1) the CLO chips forming the transformation pipeline, (2) logic to generate the two-phase clock that drives the chips from the 14.769-MHz system clock, (3) logic to handle the DMA byte transfer onto and off the board, and (4) address decoding for access to control registers and the chip lookup tables. Figure 2 displays the board architecture. An on-board DIP switch sets the base of the 2K address space of the processor.

Two channels of the PC's 8237 DMA controller are used in the single transfer mode to handle the processor I/O, with the output bytes lagging the input bytes by the total length of the pipeline (12 × 515 bits). Thus, the returned array can be targeted to the same memory block as the source array, if desired. This mode is handy for looping the pipeline since the DMA continuously resets itself automatically after 8192 transfers, and no software intervention is required in the loop to reset the DMA base addresses.

Parallel-serial interfaces are used at the input to the first chip and at the output of the last chip to interface to the DMA register on the board. After the software sends a Sync signal to the board to start the pipeline, the CPU is halted, and an Interrupt 7 request is issued by the CLO board to restart the CPU after the last DMA output. In this manner, only the

memory refresh cycle is active and can interfere with the DMA transfers, allowing them to proceed with maximum speed since this is the limiting factor on pipeline speed.

Benchmarking the performance

To benchmark the processor's power in image processing, we applied the processor to the Abingdon Cross benchmark case shown in Figure 3, which is also a nice example of some of the power of cellular logic operations. Preston at Carnegie Mellon proposed this benchmark as a processing standard. It has been used to formally test a number of existing and hypothetical parallel machines, with the results reported in Uhr et al.[10] The test involves the application of a series of standard, well-defined CLO transformations to first filter the image for noise, sharpen the feature edges, and then skeletonize the cross. The final result is shown in Figure 4.

Noise filtering by CLO transformations can be understood in terms of a simple example: the "grow" and "shrink" operations on a blob of 1's in a background of 0's. We perform these operations by specifying neighborhood rules that apply only to elements on the edge of a blob and have the effect of either eating inward by one cell or expanding outward by one cell. If a blob is large enough, a number of shrink steps followed by an equal number of grow steps will have no effect. However, a small blob of noise, say an isolated 1-cell, will disappear during the shrink steps and thus be eliminated.

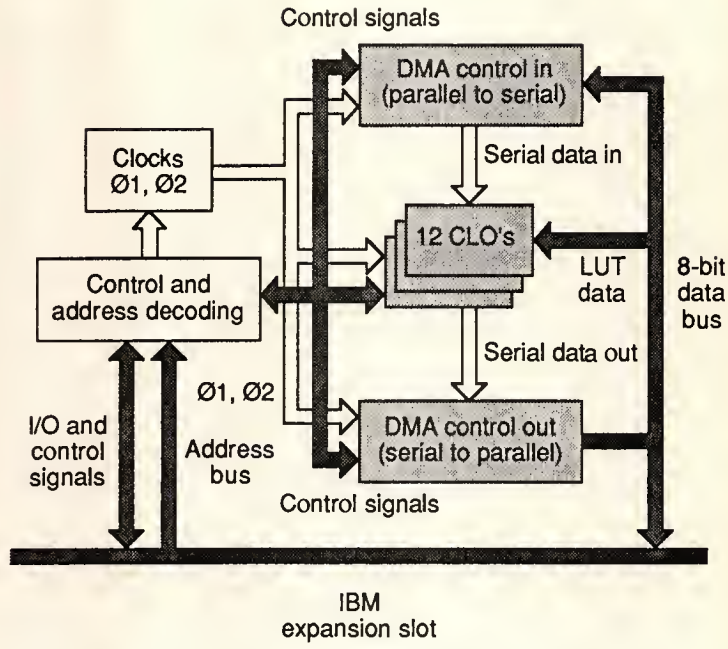


Figure 2. Processor board architecture.

Skeletonization uses a transformation similar to the shrink operation to erode the edges of any solid object inward symmetrically until a single line (or backbone) is produced.

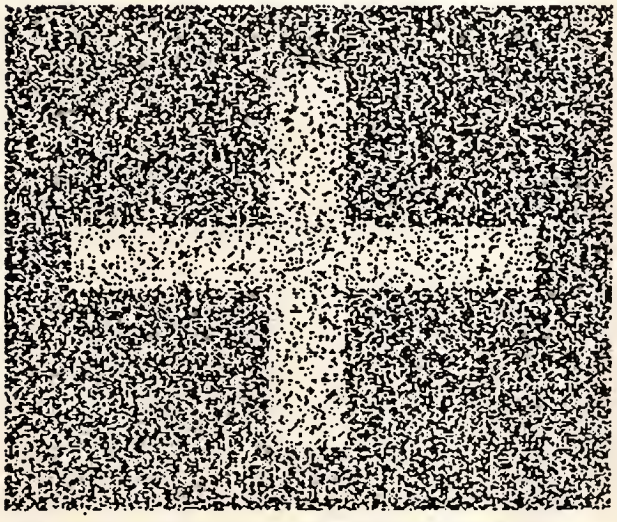


Figure 3. The Abingdon Cross benchmark image.

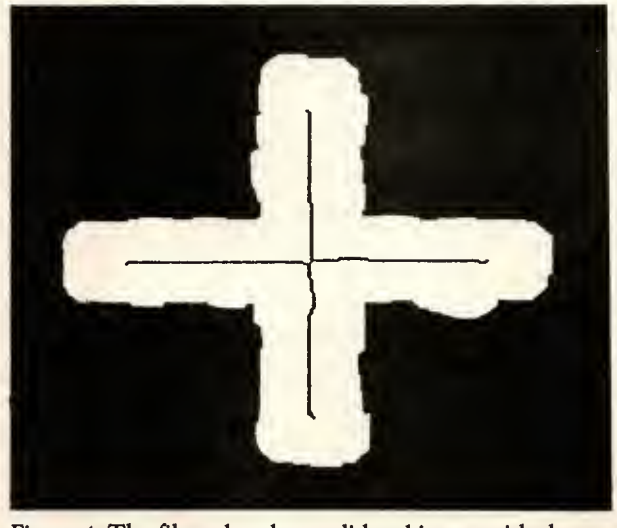


Figure 4. The filtered and consolidated image with the final skeleton shown superimposed.

Table 1.
Some systems benchmarked with the Abingdon Cross.[10]

Machine	Affiliation	Quality factor
CLIP4	University College London	7273
CYTO II	Environmental Research Institute of Michigan	39#
CYTO III	Environmental Research Institute of Michigan	102#
diff4	Coulter Electronics	17857 +#
DAP	International Comp. Ltd.	64000 +
DIP	Univ. Delft	46
FLIP	FIM/Karlsruhe	75
IP8500	Gould	†
IP8500	ETH/Zurich	78
Mach.Vision 2000	Machine Vision International	1924#
MPP	NASA Goddard	26283 +
PHP	Carnegie Mellon	7/31#
PICAP	Linkoping University	26
POP II	Royal Holloway College	2
System 75	International Imaging System	†
TAS	Leitz Wetzlar	533
TOSPICS II	Toshiba	206
TRAPIX 5500	Recog. Concepts, Inc.	40
VAP	University Berne	49
VICOM 10	Vicom	18
VICOM/CYTO	Vicom	6826 +
WARP	Carnegie Mellon	65*
CLO coprocessor	Johns Hopkins	175**

\# Algorithm used employs a priori information on width of cross.
\+ Estimated quality factor; benchmark not actually run.
† Investigator tried benchmark but could not obtain skeleton.
* Hypothetical system; benchmark estimated.
** Not included in the original Uhr et al.[10] table but added here for comparison.

The Abingdon Cross benchmark quality factor is defined to be the array size being processed divided by the clock-on-the-wall processing time for completion. The processing time should include all software intervention required to control the processor and its I/O. In our case the quality factor was about 175, based on an array size of 256 and a processing time of 1.46 seconds.

Table 1 shows some selected benchmark results from Uhr et al., all of which are for machines larger and more expensive than the CLO processor reported here. It is clear that the processor compares favorably to most larger machines for the computational task for which it was designed. For this project, the important question about performance deals with the comparison to a software implementation on a general-purpose computer. Performing 240 complete CLO array transformations per second on a 256 × 256 array requires approximately 15.7×10^7 individual element transformations per second. For implementation in software, if one assumes that only five CPU operations are involved for each individual transformation in the inner loop, this rate is equivalent to an effective processing rate of about 75 million operations per second. This performance at least equals the performance of most of the older vector mainframes on a well-vectorized problem. As a specific comparison to a software implementation, a C-language program on a VAX 11/780 produced about 0.9 array transformations per second of CPU time for this size array.

We feel a key element in the usefulness of the application-specific approach is developing the hardware as a coprocessor hosted by a personal computer. This approach is the fastest and most cost-effective way to achieve a stand-alone interactive processor. The host solves all the problems associated with interacting with the processor.

Another important element is that the processor is strongly decoupled from the software on the host machine by the use of simple block DMA transfers for processor data input and output. One benefit of this design decision is that the software required to support the processor is very uncomplicated. The few control signals needed (reset, border control, and start) are supplied by the user programs via memory writes to some absolute locations in the host computer's address space. The lookup tables for programming the neighborhood rules are handled the same way. High-level-language routines in the user's choice of language handle these functions easily.

Another advantage of the simple interface between host software and the processor is that maximum flexibility is achieved in the use of the processor. We discovered several related, but unanticipated, cellular automaton problems for which the coprocessor was able to be used. All of these resulted from the restriction of the processor to a very straightforward computing task and from the simple DMA interface used to move the data through the processor. The host machine's software assumes all the burden of managing the data's source and destination.

One example of such an unanticipated use is the case of a one-dimensional automaton array, which is an interesting computation in its own right.[6] One-dimensional cases can be handled by letting the initial conditions become the first row of a zero 2D array and setting the target base address of the DMA controller to lag the source address by one row (32 bytes). By scrolling the source address row by row

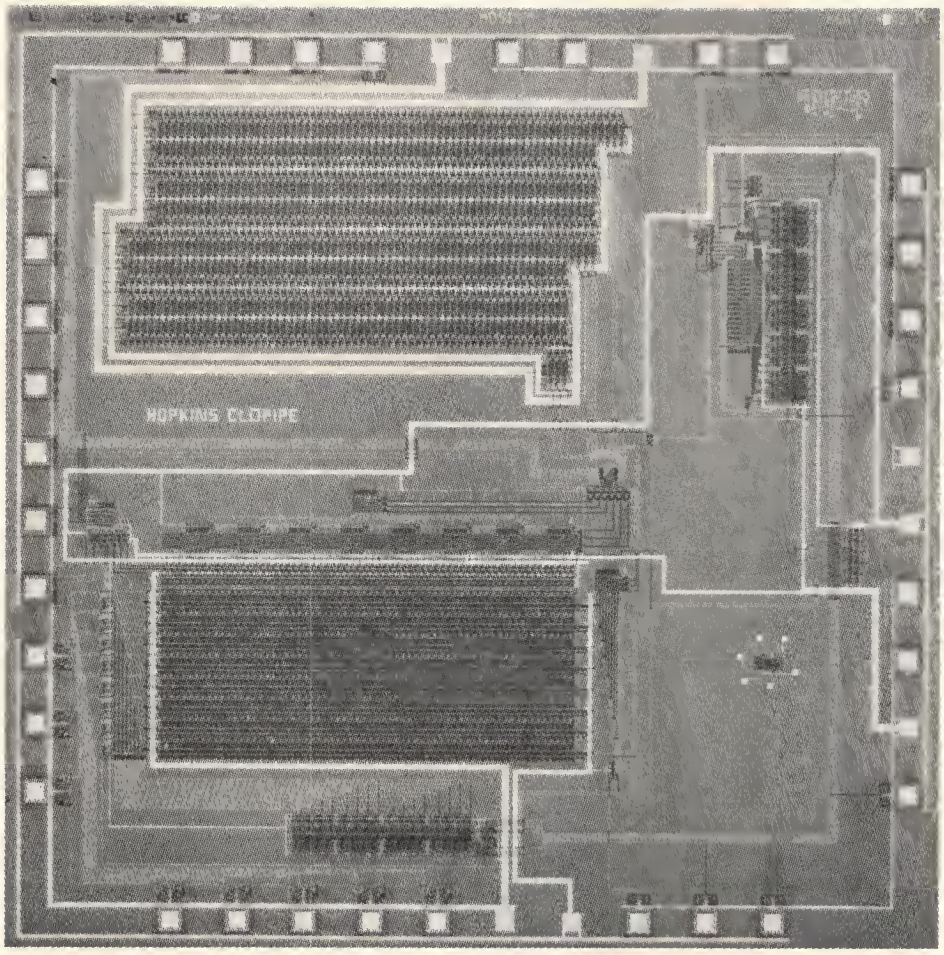


Figure 5. CLO chip laid out and simulated by students in the second semester of a VLSI design course, using the "university consortium" Unix-based CAD tools. The two major structures on the chip are the 64-byte RAM lookup table and a serpentine 515-bit shift register. The 7000 NMOS gates used in the design fit into a MOSIS-standard, 40-pin frame with obvious room to spare.

(modulo 256), one can graphically display a time history by rows, which is the usual method.[6] The software intervention required to reset the DMA base addresses each time the pipeline is called adds a small overhead in this case.

Another example is the class of transformations that are second order in time and satisfy time reversibility.[11] These transformations require that the transformation to a new state at $t+1$ be based on the two previous states at t and $t-1$. This requirement was handled by cycling the DMA source and target arrays through three separate arrays, thereby saving two predecessors of each new state. Through software intervention between pipeline calls, the correction for the earliest predecessor can be made. For two-state automata, such a procedure amounts to a simple exclusive OR of the two arrays, which can be handled by assembly code in 28 ms. The biggest throughput reduction in this case is achieved by the fact that only one transformation at a time can be handled by the pipeline, with the remaining 11 loaded with the identity transformation. Even so, we were able to watch graphics displays of the time-reversal rule referred to as Q2R in Vichniac[11] at a rate of 16 steps per second, which is definitely an interesting experience.

We estimate the total development time for the CLO chip (Figure 5) and the processor to be about six man-months. We felt this was the disappointing aspect of the project experience. One man-month seems more appropriate for the claim of nearly equivalent efforts in the hardware and the software. Clearly we have not quite reached our goal of truly fast-turnaround coprocessors, however the development time was short enough to give encouraging signals. Things are sure to improve in this regard; for example, we used traditional CAD tools for the chip layout rather than a silicon compiler that should reduce chip design time considerably. Also, the testing time required for the processor greatly exceeded the typical requirements for debugging equivalent software. More automated hardware testing techniques will surely improve this aspect within a few years.

On the other hand, for a computational task that will be run in production mode for a long period, a six-man-month development is a reasonable price for such a clear improvement in performance and throughput. For a certain class of problems in which the scope of the computing task is small and performance rather than generality is needed, the approach of using an application-specific coprocessor appears

preferable to implementation on a general-purpose machine. Even on the newly emerging highly parallel machines, the software effort involved might well be spent in producing a special-purpose processor. It is also not clear in many systems problems that are solved with an embedded microprocessor whether one might be replacing a hardware effort with an equivalent software effort and an attendant loss in performance.

Acknowledgments

Some of the architectural ideas for this processor were contributed by R.C. Moore and K. Strohbehn of the JHU/Applied Physics Laboratory. In addition the students in EE52.646 at JHU put some hard work into the CLO chip design during the spring semester of 1985. An NSF grant supported the chip fabrication.

References

1. K. Preston et al., "Basics of Cellular Logic with Applications in Medical Image Processing," *Proc. IEEE,* 1979, p. 826.
2. K. Preston, "Some Notes on Cellular Logic Operators," *IEEE Trans. Pattern Analysis and Machine Intelligence,* July 1981, p. 476.
3. M. Gardner, *Scientific American,* Oct. 1970, pp. 120-123.
4. L.S. Schulman and P.E. Seidan, "Statistical Mechanics of a Dynamical System Based on Conway's Game of Life," *J. Stat. Phys.,* 1978, p. 293.
5. C. Burks and D. Farmer, "Towards Modeling DNA Sequences as Automata," *Physica 10D,* 1984, pp. 157-167.
6. D. Farmer et al., *Cellular Automata—Proceedings of an Interdisciplinary Workshop, Los Alamos, N.M., 1983,* North-Holland Physics Publishing, Amsterdam, 1984, 245 pp.
7. K. Preston and M. Duff, *Modern Cellular Automata Theory and Applications,* Plenum Press, New York, 1984, 340 pp.
8. T. Toffoli, "A High Performance Cellular-Automaton Machine," *Physica 10D,* 1984, p. 195. (Also included in Reference 6.)
9. M.J.E. Golay, "Hexagonal Parallel Pattern Transformations," *IEEE Trans. Comput.,* 1969, pp. 733-740.
10. L. Uhr, K. Preston, S. Leviality, and J.B. Duff, *Evaluation of Multicomputers for Image Processing,* Academic Press, Orlando, Fla., 1986.
11. G. Vichniac, "Simulating Physics with Cellular Automata," *Physica 10D,* 1984, pg. 96. (Also included in Reference 6.)

Robert E. Jenkins developed the CLO chip as part of a course in VLSI design that he introduced as a visiting professor at Johns Hopkins Homewood campus in 1985. He is a lecturer in the Hopkins School of Engineering and a principal staff engineer at the Hopkins Applied Physics Lab. With this lab he has over 20 years of experience in computing systems applications and is currently leading the Space Department IR&D program.

A member of the IEEE, Jenkins received his BS in engineering and his MS in physics from the University of Maryland.

D. Gilbert Lee, Jr., performed the major portion of the processor board design and automated chip testing as part of an investigation of fast-turnaround, computer-aided design techniques. A senior staff engineer at the Hopkins Applied Physics Lab, he has experience in spacecraft digital subsystem development and is currently playing a leading role in applications of CAD tools.

A member of the IEEE, Lee received his BS in biomedical and electrical engineering from Duke University and his MS from Marquette University.

Questions concerning this article can be directed to Robert E. Jenkins, Johns Hopkins University, Applied Physics Laboratory, Johns Hopkins Road, Laurel, MD 20707.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 156 Medium 157 High 158

Software Design for Real-Time Multiprocessor VMEbus Systems

Walter S. Heath

Taming real-time systems can be a challenge. You'll gain flexibility and cut debugging time by adding additional processors and the right software.

One can get into a lively discussion on the precise meaning of the term *real time*. Those of us who design and build real-time systems understand perfectly the meaning of the term within the context of our own work. It is quite another matter to come to a consensus on a general or global definition. Indeed, even within the industry we have no agreement on the correct spelling of the term (that is, real time versus realtime)!

Perhaps it is appropriate that the term used to describe these systems cannot be defined precisely, since a primary characteristic of real-time systems is that they must deal with uncertainty. Although a precise, quantitative definition of the term does not seem to be attainable, these systems do possess some common qualitative attributes.

Generally speaking, real-time systems are *reactive* in the sense that they are driven by external events. External equipment requests (or demands) the attention of the system, and the system must respond by performing a service within some prescribed response time. The interrupt mechanism is used to signal a request for attention, and so these systems are also classified as *interrupt driven*.

In this environment the software program must be designed to cope with uncertainty. That is, it must be capable of providing services to several external events that are occurring asynchronously and to do so within the response-time constraints of each external device individually. The program is therefore nondeterministic in that it is not possible to predict exactly what it will be doing a given number of clock cycles after initiation.

One has some limited control over the order in which services are provided by specifying interrupt and task priorities. Designers may assign highest priority to the interrupts from the most time-critical external devices and thereby ensure that they will be serviced as fast as the computer system is capable of responding. Similarly, they may assign scheduling priorities to application tasks. Beyond this, the designers must base their designs on a statistical average or worst-case analysis of system-response requirements. Since this analysis is, by nature, imprecise, the success of the design must rely heavily on the previous experience of the designers involved.

A discussion of the definition of the term *real time* often degenerates into a debate over what is meant by *immediate response*. This quantitative side of the definition is the most context dependent. For purposes of this article I define the phrase response time as the time interval between the occurrence of an interrupt by an external device and the first I/O operation performed by the computer in response to that interrupt. Systems implemented in custom hardware (perhaps controlled by microcode) may have response times in microseconds or less. On the other

0272-1732/87/1000-0071$01.00 © 1987 IEEE

hand, several minicomputer systems running large operating systems have a real-time mode. Response times here are usually in milliseconds. At the outer extreme are systems that respond in human response times (that is, fractions of seconds).

It is entirely possible for a single system to require different response times at different stages in the system. An example would be a system that must gather large quantities of raw data at high speed, process the data, and present results to an operator such that the operator has time to evaluate and possibly alter system operation in a timely manner. An air traffic control radar system is a good specific example. Here large quantities of A/D samples might be collected by high-speed, dedicated hardware under the control of a slower but more versatile microprocessor system. The data processing and display operations can require the computational power of a larger mini- or mainframe computer.

In this type of system the processing requirements change as data progresses through the system, beginning with a high-speed, dedicated (that is, dumb) front end and ending with the slowest but (presumably) most intelligent "processor"—the operator. The intermediate microprocessor stage may be neither fast enough to gather the raw data nor sufficiently powerful to perform data analysis. But it may possess adequate time response and versatility to control data collection and equipment configuration. As such, it serves as the central "facilitator" of operations by directing the operation of the dedicated front end and delivering blocks of data to the back-end computer.

In this article I discuss software design and development topics for microprocessor-based systems using multiple processors. Such systems currently have response times in the tens to hundreds of microseconds. As such, they may be capable of providing the entire processing component of a system, or they may serve as a stage in a more demanding system, as described earlier. I discuss system design considerations only with respect to their influence on the software. Details of specific software components appear in the accompanying boxes.

Serial versus parallel operation

The need to be capable of responding to several asynchronous events at essentially the same time has been a challenge for the traditional von Neumann single instruction stream, single data stream (SISD) or single-thread computer. The design requirement is inherently parallel while the computer is inherently serial. The traditional solution has been to provide a serial-to-parallel "transformer" in the form of a system executive that makes the computer appear to the outside world to be operating in parallel. The executive does this by time-sharing the computer's central processing unit between multiple tasks or processes. This so-called multiprocess or multitask operation is accomplished either by providing each process with a slice of CPU time, based on some priority/round-robin scheduling algorithm, or by allowing application tasks and interrupt handlers to dynamically schedule tasks to be run by a priority task scheduler. While this approach has been very successful and has been used to implement many real-time systems over the past several decades, it has some serious shortcomings, both technical and procedural.

Multitasking by a single CPU requires system overhead to transfer control from task to task. The entire state (registers, flags, and so on) must be saved for the task being suspended, the task scheduler must find another task to run, and the state of that task must be restored. This "context switch" can take tens to hundreds of microseconds.

Another problem with SISD systems is that, as new tasks are added to the system or as existing tasks are expanded, the performance of the entire system progressively degrades. This aspect has made designing such systems a risky enterprise, since one is not able to determine overall system performance until all components are functioning in their final forms. If the final system operates too slowly, the implementers must either produce more efficient code, remove less essential components, or purchase a faster version of the computer (if one is available).

The SISD approach also has some procedural difficulties. Since all software components must run together, a considerable coordination problem usually arises during the programming phase. For example, when parameters in a common data area must be changed, all programs that access that common area must be recompiled. It is also difficult to perform system integration, since the entire program must usually be run to test each component. While these problems can and have been surmounted in the past, current technology provides another approach that largely avoids them and provides additional capabilities as well.

The multiprocessor approach

A multiprocessor architecture matches real-time design requirements more closely. Multiple events can be serviced by multiple processors running in parallel. These systems are classified as multiple instruction stream, multiple data stream (MIMD) or multithread computers. Since less task switching takes place in each processor, executive overhead is reduced. The computational load is also distributed among several processors, thus relaxing individual processor execution time constraints.

The software in multiprocessor systems is often partitioned functionally; processors can be dedicated to performing specific parts of the job. An important

by-product of partitioning is that each individual processor program is simpler, is often written by one person, and can be tested independently. Thus processor functional partitioning leads naturally to the partitioning of software development responsibilities among personnel. Since programs can be written and tested independently, these efforts can proceed in parallel.

In a multiprocessor system a formal means for interprocessor communication must be established. Important system considerations in choosing the communication medium are:

1) hardware and software overhead to access the medium and
2) the data transfer rate after access is granted.

The MIMD class of computers can be further subdivided at this point into tightly coupled and loosely coupled subclasses. For systems in which high volumes of data must be shared between multiple processors with minimum latency, processors must be tightly coupled either by means of direct data paths or via a common multiport memory. For systems with less severe communications requirements a less tightly coupled and more flexible arrangement can be used.

Several media and communication protocols have been standardized for coupling processors in a loosely coupled configuration. For systems in which some communication speed may be traded for design flexibility, the backplane bus technologies are appropriate. Of these, the VMEbus has gained favor for many real-time applications.[1-4]

From the system designer's point of view an important feature of the VMEbus is its nonproprietary, open architecture. As a result, many suppliers are currently producing a wide spectrum of single-board computer (SBC) and I/O device interface boards. A short time after a new data or signal processing chip is introduced, it appears on a VMEbus board. It is therefore possible to configure the computer to match system requirements, rather than the other way around as was done in the past.

The VMEbus provides a high-speed, common interconnection between multiple SBCs and I/O devices. VLSI interface chips handle bus access protocol, so these operations are largely transparent to the software. The bus provides flexible priority interrupt and priority bus access request/grant mechanisms. I/O device ports and SBC dual-ported memory can be mapped into a single linear address space. One SBC can access the memory of another by simply generating addresses in its memory range; the SBC accesses I/O device ports as memory locations. Finally, blocks of data can be transferred across the bus at high speed using DMA devices, so that, on the average, the percentage of bus bandwidth being used can be kept low.

Passing Message Pointers

A previous *IEEE Micro* article describes a set of functions for passing messages among application tasks and interrupt handlers using queues.[5] For some applications it is preferable to pass pointers to messages, rather than the messages themselves. This is particularly true when the messages are large. Passing a pointer takes considerably less time than passing an entire long message. Since pointers are fixed in length, the queues have fixed-size entries, and so the queue access functions are simpler.

A buffer pointer queue consists of a queue header and an array for storing pointers to buffers. The header has the following structure:

```
struct pque{          /*buffer ptr queue header  */
      char sema      ;/*access semaphore         */
      char full      ;/*queue-full flag           */
      short head     ;/*head pointer (index)      */
      short tail     ;/*tail pointer (index)      */
      short lngth    ;/*queue length (# of ptrs)  */
      short count    ;/*count of msgs in queue    */
      char task      ;/*waiting task ID           */
      long *pbuf     ;/*ptr to ptr queue array    */
}                    ;
```

A queue's *head* pointer (index into the pointer array) points to the next buffer pointer to be removed. A queue's *tail* pointer points to the next open entry in the queue. Note that these pointers will end up pointing to the same entry when the queue is either full or empty! The Full flag is used to resolve this ambiguity. The pointer queue's length is specified by *lngth*. This flag is used to determine when to *wrap* a head or tail pointer (that is, move it from the end to the start of the pointer array).

Actual access to a queue is performed by primitive functions Putbp() and Getbp() (put/get a buffer pointer), so the user does not need to manipulate parameters in the queue header directly. If these functions are unsuccessful (the queue is full or empty), they return -1. Various higher level functions that supply useful additional services call these primitive functions. For example, functions Putbpwt() and Getbpwt() (put/get a pointer or wait) use a queue header variable *task* to determine if another task attempted to access the queue and was unsuccessful and therefore suspended action (called Sleep().) If a number other than minus one is present in *task*, it is interpreted as the ID of a suspended task and that task is awakened.

Thus, if Putbpwt() places a pointer in a queue and finds that another task has called Getbpwt() in an attempt to get a pointer and found none was available, it will wake the suspended task. Since a pointer is now available, that task can proceed. Similarly, if Putbpwt() attempts to place a pointer in a full queue, it will suspend. Then, when Getbpwt()

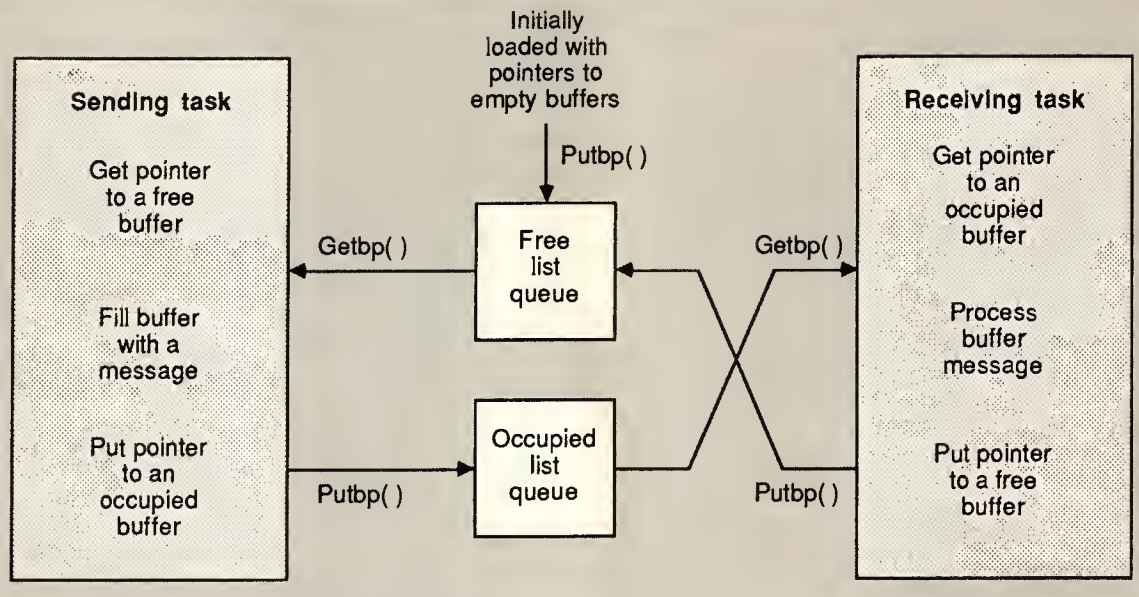


Figure A. Procedure for using buffer pointer queues.

removes a pointer, it will wake the waiting task. These two functions therefore provide a mechanism that allows data flow to control task execution.

Obviously, the pointer to the array of pointers *pbuf* and all of the other header variables must be initialized before they can be used. An initialization function performs this duty as part of the executive's start-up procedure.

The procedure for using buffer pointers in an application is diagrammed in Figure A. For a particular intertask (or task-interrupt handler) path, a set of two buffer pointer queues is defined, one a *free list* and the other an *occupied list*. The free list queue is initially loaded with pointers to fixed-size message buffers, using Putbp(). Then, when a task needs to send a message, it calls Getbp() (or a higher level calling function) to remove a pointer to a free buffer from the free list queue, loads the buffer with data, and then places the buffer's pointer in the occupied list queue, using Putbp(). The receiving task, when it needs a message, calls Getbp() to remove a message pointer from the occupied list, processes the message, and returns the pointer to the free list, using Putbp().

Note that the pointer queue functions are not concerned with the contents of the messages passed. Obviously the application program must be aware of message format and size. If message size is variable, it can be sent as part of the message. It is also clear that a set of fixed-size buffers that are as large as the largest message sent must be defined for each message path. The maximum number of messages that can be queued at any one time is also fixed. Finally, the execution time required to pass access to a message is fixed, so execution time is independent of message size.

By contrast, when messages themselves are passed in the queues, the number of messages that can be queued depends on the number that can be placed in the queue array. A large number of small messages can be queued or a smaller number of larger messages can be stored. In addition, the same queue space is used repeatedly as messages are passed to and from the queue. These operations therefore use memory space more efficiently. But the time required to pass a message is directly proportional to its length.

When passing message pointers between SBCs, a semaphore flag must be used to control access to the queue. Otherwise two SBCs might attempt to manipulate a queue header at the same time. The *sema* flag controls access. A special assembly language instruction must be used to manipulate this flag. For example, the 68000's *tas* (test and set) instruction indivisibly tests and sets the flag. The instruction returns the setting of the flag previous to instruction execution (in the processor's Z status bit). By convention, when sema is set, access to the queue is blocked. Thus, when tas is executed, it blocks access and returns the previous access condition. If the queue was not already blocked (sema was cleared), the calling SBC can then access the queue. If the queue was blocked, it must wait. When an SBC is finished with a queue, it must clear sema to allow access by the other SBC. An indivisible instruction is required to avoid the situation where both SBCs test the flag, find it cleared, set it, and then attempt to access the queue at the same time.

Functions Putbpr() and Getbpr() (put/get buffer pointer or retry) provide inter-SBC queue operations. These functions test the semaphore flag before calling Putbp() or Getbp(), respectively. A calling argument specifies the number of times the semaphore flag should be polled before failure is reported (-2 returned).

System design considerations

Any common bus system should be designed to minimize bus traffic. Although VMEbus data throughput is high, the bus is usually the only data channel between the SBCs and is therefore a potential bottleneck. Bus traffic can be reduced by passing message buffer pointers rather than the messages themselves. When the final destination for the message has been determined, the message can be passed at high speed using a DMA device. Wherever possible, I/O should be performed directly from the SBC board using on-board I/O devices. Several SBC boards contain serial and parallel ports and disk controllers that perform I/O either via the board's P2 connector or through the front panel. The use of these ports removes some of the traffic from the bus.

Another way to reduce system design risk is to simply design in computational margins at the beginning. Since SBCs are relatively inexpensive, it is not unreasonable to use more of them than the initial design requires so that each one operates below its potential capacity. Since smaller programs in more SBCs are easier to write and test than larger and fewer programs, the extra cost of additional SBCs will usually be offset by lower software production costs. In addition, since timing constraints in each SBC are relaxed, programs can be written in a high-level language; the associated execution time penalty is usually of little concern.

It is important that system integration and debug procedures be considered during the initial design phase. Individual SBC programs can be tested using the debugger in the SBC's PROM monitor or perhaps by using an in-circuit emulator. It may be necessary to write separate source and destination simulator programs and run them in adjacent SBCs to simulate bus traffic to and from the program being tested. These can be simple, menu-driven loop programs.

It is also important to take into account the types of data processing that will be performed. Common types of operations can then be grouped together in dedicated SBCs. For example, operations can often be partitioned into computation-intensive, communication-intensive, and I/O-intensive categories.

A typical design

Figure 1 depicts a candidate real-time VMEbus system. It contains a bus arbiter board and several SBC and I/O boards. A design might contain a central dispatcher SBC that collects message pointers from other SBCs, determines message types, and dispatches them to appropriate destination processors. Since this processor is the center of activity, it might also support data transfers for the system's most time-critical I/O devices. For example, it might control message traffic to and from a high-speed communications channel.

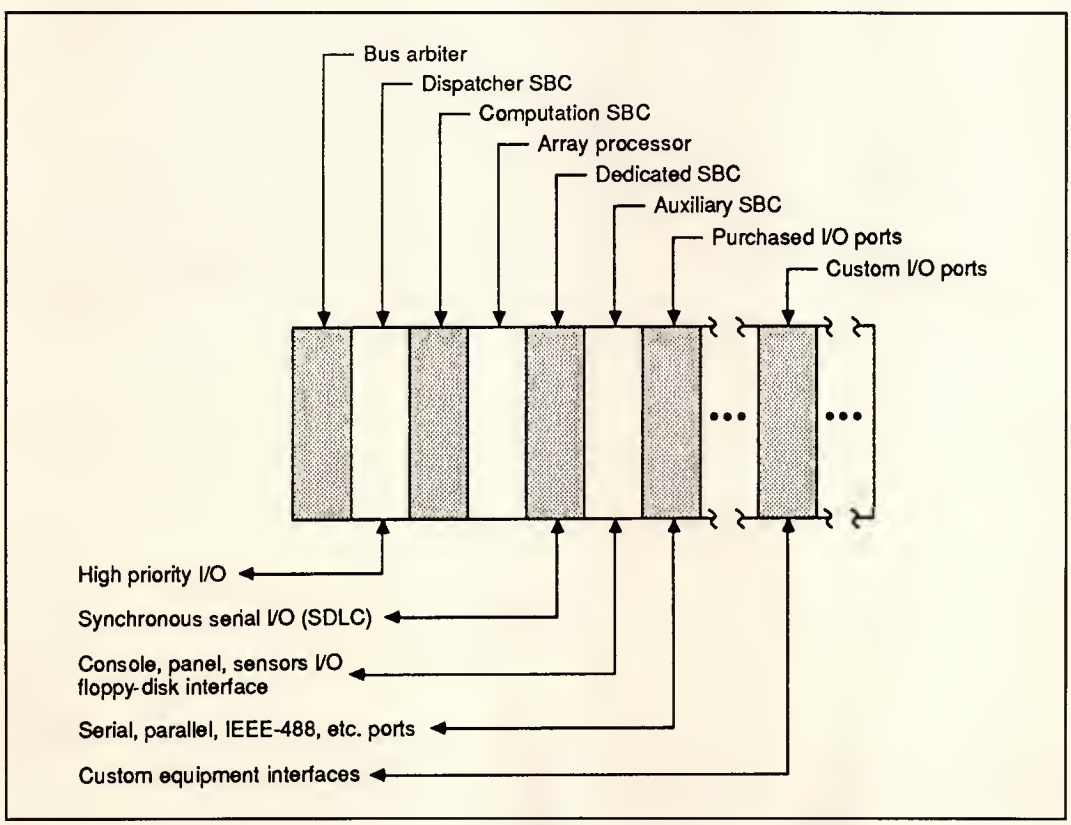


Figure 1. Typical real-time VMEbus system components.

Communicating Between SBCs

In a multiprocessor system designers must establish a formal procedure for communicating between processors. Since each SBC's dual-ported memory is separately mapped into the common address space of the system, it is possible to establish common data areas for communication. C language programming techniques that can be used to establish and access these commons follow.

A data structure type can be declared that contains all variables and data structures to be placed in the common area. An example structure appears in Figure B.

In this example I have established a set of buffer pointer queues between source and destination SBCs. The example also includes parameters for reporting status and error conditions (an auxiliary SBC might collect this data and report it on the operator's console). Finally, the Ready and Go flags used during program start-up appear (see main article).

In the source and destination SBC programs a pointer to a structure of type sbccom can be declared and initialized as follows:

```
#define SBCSTART  0x400000
struct sbccom *sp                ;
sp = (struct sbccom *)SBCSTART   ;
```

In this example the pointer is declared to point to absolute hex address 0x400000, which may be either an address of memory on the source or a destination SBC board (or on another board). Since sp points to this address and is declared as a pointer to structure sbccom, the sbccom data structure overlays a memory area starting at the pointer address. It is then possible for source and destination programs to access parameters in the sbccom structure using pointer sp. For example, the source status can be initialized to zero as follows:

```
sp->srcstat = 0                  ;
```

Similarly, message buffer pointer msgptr can be placed in queue osrcdst using function Putbpr() as follows:

```
putbpr(msgptr,&(sp->osrcdst),1,1)   ;
```

As noted in the box titled Passing Message Pointers, function Putbpr() uses a semaphore to control access to the queue. Since the VMEbus supports indivisible instruction execution (the 68000's tas instruction), it is valid to use the semaphore technique to control access to queues that are accessed by multiple SBCs.

This procedure for passing parameters and buffer pointers can be used to establish common areas between the various SBCs in the system. By using several common areas, less recompilation is necessary when parameters are added or changed. Structure declarations such as sbccom should be placed in separate include (.h) files in a common library so that the current version is always included when programs are recompiled.

Note that a process must poll a queue to gain access to it and then to determine whether a message pointer is present or whether room exists to store a pointer (the last two arguments in the Putbpr() call specify the number of times the function should poll before returning failure). Experience has shown that this is not a serious problem. If an SBC is busy processing a previously received message, it should probably not be interrupted; if the SBC is not busy, it might as well spend its time polling its queues. Queues that can be expected to require substantial polling should be placed in the memory of the SBC that will be doing the polling so that excessive polling will not occur across the bus.

```
struct sbccom{
    struct pque fsrcdst          ;/*free list queue header     */
    int fsdbuf [FSDLNGHT]        ;/*free list ptr buffer        */
    struct pque osrcdst          ;/*occupied list que header */
    int osdbuf [OSDLNGTH]        ;/*occupied list ptr buffer    */
    int srcstat                  ;/*source status               */
    short srcerr1                ;/*source error #1 flag        */
    short ready                  ;/*SBC-ready flag              */
    short go                     ;/*SBC-go flag                 */
}                                ;
```

Figure B. Sample data structure.

Most systems will also need an auxiliary SBC to handle lower speed interfaces, such as the operator console, control panel interfaces, and/or environmental sensors (temperature, position, vibration). One or more SBCs might be dedicated to performing computation-intensive operations. These processors might contain conventional CPUs with attached coprocessor floating-point math chips, or they may be array processors. In some designs SBCs might be dedicated to supporting specific I/O operations. To get maximum performance, certain high-speed I/O devices require the undivided attention of a separate processor (a high-speed, synchronous serial communication link).

A typical system will also contain several custom and/or purchased I/O boards. Boards for most standard I/O protocols are available. Special-purpose interfaces can be fabricated using wirewrap techniques. In all cases the software interfaces via memory-mapped registers and interrupts. Some boards include DMA chips to support high-speed I/O transfers.

The system executive

The software that runs in the SBCs can be divided into two general categories, system software and application functions. A system executive provides an environment within which application programs can be constructed and run. As with governments, a system executive should be as small as possible—consistent with the need to supply essential services. An executive should not waste machine cycles on unnecessary or inefficient operations. A cycle lost to system software is a double loss to the application since both a cycle of useful computation and a cycle of real time are lost. For high-speed systems the executive should therefore be designed to meet the specific needs of the application as closely as possible. It should also provide a convenient interface to application programs and should not be so complex as to be difficult for users to understand and use.

At minimum, an acceptable system executive should provide priority task scheduling, support interrupt processing, and provide formal mechanisms for intertask and interprocessor communications. But, some of the services provided by even this small set of functions may not be needed in some applications. For example, if system requirements dictate that each SBC in a multiprocessor system should run only one task, a task scheduler is not needed. In systems that must provide the highest possible performance, unnecessary executive components should not be included.

One must decide whether to purchase an executive or write one. A custom executive can be designed to meet the specific needs of a project and can be modified, if necessary, as application requirements change. If the source code for the executive is avail-

The System Executive and Preemptive Scheduling

The system executive design described in a previous *IEEE Micro* article provided priority, nonpreemptive task scheduling, and utility functions for passing entire messages using queues.[5] Since then, the executive has been extended to include functions to pass message pointers between tasks and between processors. Here, I briefly review the work and present some thoughts on preemptive scheduling.

Figure C presents a structural overview of the executive. It operates as follows. The user must define a Task Control Block (TCB) data structure and a stack for each application task. TCBs are linked in a list, with the last TCB linked to the first, to form a loop. The scheduler function Sched() cycles through TCBs looking for a task that has been awakened. The executive spends its time in this way if no tasks have been awakened.

An interrupt handler or another task can wake a task by calling function Wake(). When Sched() finds an awakened task, it calls Run(). Function Run() saves Sched()'s stack pointer (SP) and transfers control to an entry point in Sleep() (using assembly code). Function Sleep() gets the task's SP from its TCB and returns to the task. (The return address was pushed onto the task stack during a previous

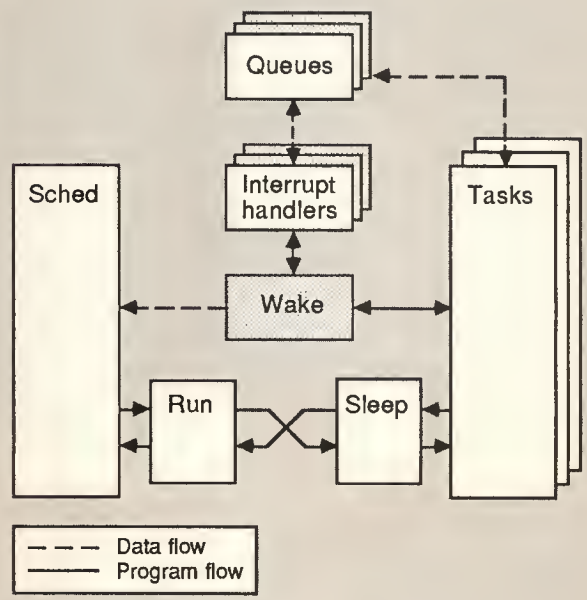


Figure C. System executive, structural overview.

call to Sleep().) The task loop performs user-supplied operations and eventually calls Sleep(). Sleep() transfers control back to Run(), and Run() returns to Sched(). The task's SP is saved, and Sched()'s SP is restored in this operation. Since Sched() always starts scanning TCBs from the top of the linked list, the order of TCBs on the list determines task priority.

When an interrupt occurs tor a high-priority device, the interrupt handler may need to cause the interrupted task to suspend action so that the data received by the handler can be processed by a higher priority task. That is, the lower priority task is preempted by the higher priority task. Note that this procedure is needed only in closed-loop feedback situations in which the received data must be processed immediately so that an output operation can be performed within some small time interval. In most real-time applications this level of performance is not needed and should not be used, since it imposes additional system overhead and application programming constraints. It also makes program debugging more difficult.

If a task is preempted when it is in the process of manipulating a data structure (for example, a queue header), that structure is left in an inconsistent state. If the preempting task then accesses the same structure and changes its contents, the preempted task has an inconsistent structure to deal with when it again runs. Clearly it will be necessary to disable preemption when certain "critical regions" of code are being executed. A similar situation arises for common functions that may be called by both tasks (C library functions, PROM utilities). The partial results from the call from the preempted task must be saved so that they are not overwritten by the call made by the preempting task. Functions that are programmed to cope with this situation are called reentrant.

To support preemptive scheduling, an executive must be capable of providing a mechanism that will cause the preempted task to suspend at the point of interruption. The executive must then reschedule the task so that the next time it is selected by the scheduler it will run from the point of interruption. That is, it must pause to allow the scheduler to run the higher priority task. Suspension can be accomplished by manipulating the data that is pushed onto the supervisor stack when the task is interrupted.

The only penalty in using nonpreemptive scheduling is that an interrupt handler may awaken a higher priority task and that task will not get a chance to run until the current task suspends. In actual systems this problem can usually be solved by inserting calls to a Pause() function at various points outside critical regions in the lower priority tasks.

Pause() simply calls Wake() to reschedule the calling task and then calls Sleep() to suspend the task. This procedure allows the scheduler to check to see if a higher priority task has been awakened and to run it if it has. When the task that called Pause() becomes highest priority, it will be resumed by a return from the Pause() call. Insertion of Pause() function calls is a fine-tuning exercise that is usually performed in the latter stages of system integration to improve overall system performance.

Another way to avoid preemptive scheduling in a system that must support fast response to some critical I/O interface is to simply devote a separate SBC to handling that device exclusively. In this approach the program running in the SBC is likely to consist of only one task. It is therefore unnecessary to preempt the task to start another. An interrupt handler can simply return without having to check to see if the running task should be preempted. This method provides the fastest possible interrupt response time.

I believe that preemptive scheduling adds unnecessary risk and complexity in all but a few designs and that ways can usually be found to avoid it in those few remaining situations. A nonpreemptive environment is more predictable. A task running in this environment maintains control of the CPU until it chooses to give up that control (except for interrupt servicing). Thus, a task can be assured that a non-reentrant function has been completed or that all items in a data structure have been updated before another task is allowed to access that function or structure. It is therefore unnecessary to suffer the overhead of locking and unlocking code in critical regions.

able, users will have a complete understanding of every detail of the SBC programs they are writing. This knowledge contributes considerably to software productivity.

I described an executive design that is appropriate for many VMEbus systems in a previous article.[5] This executive is designed to be written primarily in the C language and in assembly language, where necessary. It can be placed in a library in a development system and linked to application tasks and interrupt handlers. The original design and extensions are discussed in the accompanying boxes. This executive is "lean," and therefore fast, for an executive that is designed to be written in a high-level language.

An executive written exclusively in assembly language would be faster but would be more difficult for users to understand. In a particular application, if extreme speed were needed, certain critical regions could be coded exclusively in assembly language.

The executive design has been extended to support the passing of pointers to message buffers between tasks within a processor and between processors via the bus. Buffer pointer queues pass the pointers. It is important to avoid unnecessary transfers of blocks of data within processor programs and especially across the bus. Instead, a pointer is passed. The destination task can then examine the message header and select the next processing step. Once a decision is made on the ultimate destination for the message, the pointer can be used to make a single, high-speed block transfer. Since all dual-ported memory and I/O ports are mapped into one linear address space, it is easy to accomplish a block transfer either by means of a program loop or by a DMA device.

Initial program load and start-up

In the process of writing and debugging individual processor programs, it is convenient to be able to download memory images from the development system to the target VMEbus processor using a serial link (I discuss software development next.) But during system integration and in the final system it is more convenient to be able to automatically load programs from a floppy disk. Since each SBC's dual-ported memory is separately mapped into the common address space of the system, the SBC performing the autoload has little difficulty transferring programs to other SBCs over the bus. The PROM monitor for the autoload SBC usually must be extended to support disk operations.

The modified PROM monitor program should be capable of reading a command (script) file from the disk. This file should contain a list of names of memory image files to be loaded. It should then be capable of reading these files consecutively into the memories of the various SBCs. If the software is developed on a Unix (or PC) system, the PROM monitor can be modified so that files can be read from a Unix (or PC-DOS) directory on a floppy disk. A file will then have a standard header that will contain the program's starting address and size.

Once the programs are loaded, they can be started by means of front-panel toggle switches on the individual SBCs. Reprogramming the Abort (or other) front-panel switch to start the program (another PROM patch) accomplishes this task. Each SBC can then perform start-up initialization operations and set a Ready flag in a common memory region. When all SBCs have set their Ready flags, the auxiliary processor can set Go flags for each SBC in a sequence that is appropriate for the application. Each processor then proceeds to run its real-time program. (SBC suppliers please take note that it would be a great help to users if the PROM monitor additions mentioned here could be standardized and included in delivered PROMs.)

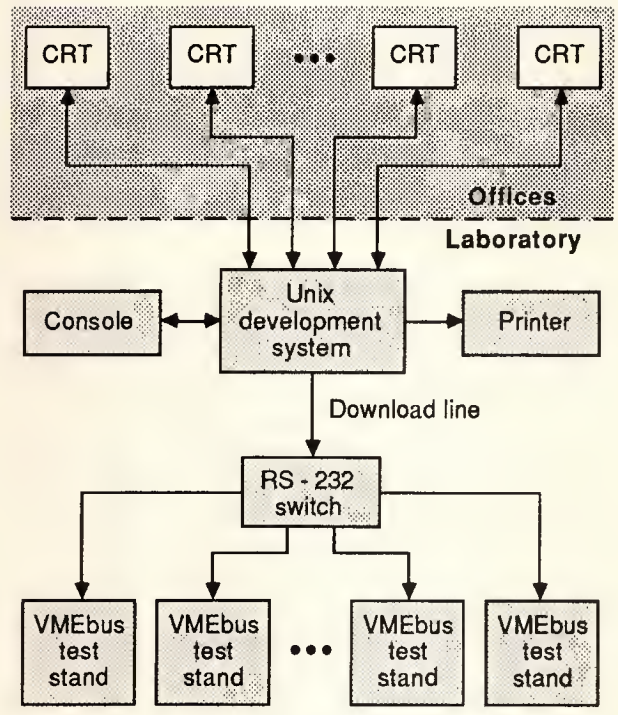


Figure 2. Software development configuration.

Software development

A good environment for developing processor programs is a small Unix system that is using the same CPU chip as the target SBC boards. The system's Unix compiler, assembler, and linker can then be used to write application programs. Otherwise a cross-compiler is needed. A Unix utility is also available for generating a memory load map, which is needed during debugging. Unlike some larger minicomputer systems with proprietary operating systems, a small Unix system can be supported by the people using it. There is no need to have a system administrator with special knowledge of the system to keep it running.

An efficient equipment configuration is shown in Figure 2. It consists of a Unix system with serial lines to user's offices for terminals, a separate VMEbus "test stand" for each user in a central laboratory location, and a serial port for downloading memory

images to the user VME systems. This download line should have a switch so that downloads can be directed to the various test stands. By having a separate test stand, each user can configure a system with the necessary SBC and I/O boards required to test software. In this way users are not delayed by waiting for access to test equipment.

At first glance the multiprocessor approach to designing real-time systems might appear to be overly complex and risky. It could be argued that a more traditional single-processor design would be safer. But experience has shown that the reverse is often true. Additional processors provide more design flexibility, more closely match the parallel nature of real-time systems, and provide more computational throughput. It is also easier to produce, debug, and integrate several simpler real-time programs than a single complex program. The inherent parallel nature of the system also means that both hardware and software development activities can proceed more smoothly in parallel. In short, multiprocessor systems reduce design risk, improve performance, and increase productivity.

Acknowledgments

This article is based on projects performed at Lincoln Laboratory, Lexington, Massachusetts. Over a dozen people contributed to these projects. The following are representative. Gary W. Ahlgren was the principal systems architect and source for solutions to complex technical problems. Tim Bowler, Marie A. Donnelly, Ellen M. Jervis, Anne M. Matlin, Linda Riehl, and the author produced software. Carol C. Martin provided valuable advice on the content and structure of this article.

This work was sponsored by the US Department of Defense. The views expressed are those of the author and do not reflect the official policy or position of the US Government.

References

1. *VMEbus Specification Manual, Rev. C.1,* Vita, 10229 North Scottsdale Road, Suite E, Scottsdale, AZ 85253; (602) 951-8866.
2. *VMEbus Systems* (magazine), 25875 Jefferson at Madison, St. Clair Shores, MI 48081; (313) 774-8180.
3. W. Fisher, "IEEE P1014—A Standard for the High-Performance VME Bus," *IEEE Micro,* Feb. 1985, pp. 31-41.
4. J. Hemenway, "Powerful VME Bus Features Ease High-Level μC Applications," *EDN,* Jan. 12, 1984, pp. 158-165.
5. W.S. Heath, "A System Executive for Real-Time Microcomputer Programs," *IEEE Micro,* June 1984, pp. 20-32.

Walter S. Heath is a computer systems designer with 20 years of experience in building real-time embedded computer systems. He has spent the last six years at Lincoln Laboratory on a contract basis building research prototype systems, including contributions to the TCAS Airborne Collision Avoidance System and a GPS navigation system. His previous experience includes the design and implementation of computer navigation, communication, and control systems including initial computer implementations of Loran and Omega radio navigation systems.

Heath received BSEE and MSEE degrees from the University of Michigan, where he specialized in electromagnetic field theory and computer technology.

Questions concerning this article can be directed to the author at 118 Fairhaven Road, Concord, MA 01742. Readers interested in further study of real-time system software can obtain the source code for the executive and example real-time application programs for 680X0 processors on a PC 5.25-inch floppy disk from Intr-Soft Co., PO Box 351, Bedford, MA 01730; (617) 369-6242.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Interest Card.

Low 168 Medium 169 High 170

1987 Index

IEEE Micro Vol. 7

This index covers all technical items that appeared in this periodical during 1987, and items from previous years that were commented upon or corrected in 1987.

The *Author Index* contains the primary entry for each item, listed under the first author's name, and cross-references from all coauthors. The *Subject Index* contains several entries for each item under appropriate subject headings, and subject cross-references.

It is always necessary to refer to the primary entry in the *Author Index* for the exact title, coauthors, and comments/corrections.

SUBJECT INDEX

+ Check author entry for coauthors

† Check author entry for subsequent corrections/comments

+ Check author entry for coauthors

† Check author entry for subsequent corrections/comments

Annual Index

AUTHOR INDEX

+ Check author entry for coauthors

† Check author entry for subsequent corrections/comments

MicroLaw

Richard H. Stern
Law Offices of Richard H. Stern
2101 L Street NW, Suite 800
Washington, DC 20037

Manufacturers' disclaimers of liability

A major US chip manufacturer put these notices on some of its application notes:

Life support applications

Alpha [a fictitious name] products are not designed for use in life support appliances, devices, or systems where malfunction of an Alpha product can reasonably be expected to result in a personal injury. Alpha's customers using or selling Alpha products for use in such applications do so at their own risk and agree to fully indemnify Alpha for any damages resulting in [sic, from?] such improper use or sale.

Life support policy

Alpha products are not for use as critical components in life support devices or systems without express written approval of an officer of Alpha Corp. As used herein:

1. Life support devices or systems are devices or systems which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can reasonably be expected to result in a significant injury to the user.

2. A critical component is any component of a life support device or system whose failure to perform can reasonably be expected to cause the failure of the life support device or system, or to affect its safety or effectiveness [sic, adversely?].

What is this all about? Does it work?

Probably, chip manufacturer Alpha is concerned that systems designer Beta will design pacemakers, drug pumps, and the like (maybe body function monitors) that use operational amplifiers, microprocessors, or gate arrays; that Beta's system will then fail in use, leaving the user dead; and that the user's heirs and/or Beta will sue Alpha for damages. At least, the second notice addresses that possibility. The first notice might be intended to cover a little more ground. Since it does not define "life support systems," Alpha might claim that airplane or automobile navigation, brake, fuel, or anticollision devices are included in the disclaimer. However, the courts usually interpret any ambiguous language in such notices against the interests of the manufacturer who wrote the notice.

Between totally innocent John and unknowing Alpha, who should bear the cost of the loss?

Alpha has a legitimate problem, but the notice probably will not offer much protection. The problem is that Alpha does not have the least idea what systems houses will do with its chips. Alpha sets a price for its chips on the assumption that it is marketing a commodity that will go into ordinary electronic devices whose failure, while definitely a nuisance, is not fatal. Alpha is not in the insurance business, whether to guarantee the business success of its customers or that of their mutual end users. Even if it somehow were willing to go into the insurance business, Alpha would not know how to set appropriate premiums.

If Alpha tried to raise its chip prices to establish a contingency fund for lawsuits, it would have two major problems: (a) Alpha would not know how much of a fund to set up (and thus how much to raise its chip prices); and (b) if Alpha raised its chip prices, rival chip manufacturers Gamma, Delta, and Epsilon would mop up the floor with Alpha. Incidentally, some of Alpha's rivals are either offshore, judgment-proof, or unsuable. The free market will not sort things out neatly for Alpha.

On the other hand, John Q. Public is out there innocently buying the Beta pacemaker that fails, or buying a ticket on Flight 006 that strays into Russian

airspace and is shot down. In either case, the system failure may be attributed to a badly designed Alpha operator amplifier. John's heirs point out that it's not his fault that Alpha made bad op amps, adding that John had received no notice from Alpha that its defective products could risk his life. Between totally innocent John and unknowing Alpha (who does not specifically foresee the exact harm to John when an imperfect op amp fails), who should bear the cost of the loss? Especially when Alpha has the deeper pockets? In fact, John's heirs might argue that the "life support" notice is evidence that Alpha knew its product might kill John; yet, Alpha went ahead and sold the product anyway. Readers will figure out for themselves how courts and juries are likely to work that one out.

Perhaps the systems house, Beta Corp., is in a different position. It was actually warned, if it ever saw the application note.[1] Because Alpha gave notice to Beta that it specifically disclaims liability for life support systems, a court would probably say that Alpha is not liable for consequential damages resulting from chip malfunction.[2] In contracts between businessmen—rather than between a businessman and a consumer—courts usually let the parties allocate the economic risks of a sale by bargaining, at least in the absence of some type of oppression. Moreover, in the sale of goods between businessmen, the assumption is that the seller will not be liable for consequential damages resulting from risks of a type or magnitude that the seller cannot foresee. By using the disclaimer, Alpha is trying to avoid a factual controversy with Beta over what Alpha should have foreseen.

What about the provision that customers like Beta "agree to fully indemnify Alpha for any damages" from sale or use? In theory, this means that Beta agrees to reimburse Alpha for any damages that John Q. Public or his heirs collect from Alpha for pacemaker failure. Probably, such a disclaimer is no more than a lawyer's theory, or whistling in the dark. A notice on an application note is not a contract, even if Beta admits to reading it. In some states, a manufacturer might be able to get away with such a disclaimer by putting it on the back of the *invoice* for the chips, but a disclaimer on an application note is probably ineffective in any state.

In contracts between businessmen, the parties usually allocate the economic risks of a sale by bargaining. The seller is not considered liable for unforeseeable damages.

All of the foregoing discussion has been directed to a chip manufacturer's notice. Readers no doubt have seen similar notices, and more sweeping ones, on documentation accompanying software.[3] Subject to limited qualification, the same principles apply to software and hardware alike. Software marketers may argue that principles like implied warranties of fitness that apply to goods do not apply to software, which is a "service" or an "intangible," but courts do not put too much stock in that argument anymore. Whether the software or the chip sends the pacemaker haywire or dispatches the airplane to Siberia, the legal problem will probably be the same.

References

1. Beta's personnel will probably deny seeing the notice. It might have been a better idea to put the notice on all of Alpha's invoices, which Beta would find harder to deny having seen (unless Beta buys indirectly from a distributor or a jobber). Certainly, Alpha cannot fit either of those notices on the chip itself.

2. A more general disclaimer might be ineffective in some states, where courts would take the position that the chip should possess good merchantable quality and be reasonably fit for its intended purpose (whatever that is). Other states might even allow a general disclaimer of any responsibility for bad chips beyond replacement cost of the chips, but again perhaps only between businessmen.

3. Provisions of this kind are typical in "shrink-wrap licenses," which are printed provisions accompanying disks, wrapped inside shrink-wrap plastic. Typically, they state that tearing open the plastic to get at the disk constitutes user "agreement" to the terms of the license.

A recent decision in Louisiana held a shrink-wrap licensing clause unenforceable. The State of Louisiana passed a law making shrink-wrap licenses enforceable. Vault (the proprietor of the Prolok copy-protection system) sued Quaid (the proprietor of the CopyWrite program) for defeating copy protection. Quaid allegedly aided consumers to unprotect third parties' programs protected with Prolok, in violation of the third parties' shrink-wrap licenses forbidding consumers to copy the programs. Quaid also disassembled Vault's Prolok to analyze it and by reverse engineering developed a part of CopyWrite to overcome Prolok. In so doing, Quaid allegedly violated a provision in Vault's Prolok shrink-wrap license that forbade disassembly. The Louisiana shrink-wrap licensing law expressly authorized shrink-wrap licensing that prohibited "translating, reverse engineering, decompiling, [and] disassembling" computer programs.

The court held that the allegedly illegal copies of third-party software were none of Vault's business: Any complaints about violation of rights in such programs should be asserted by the proprietors of the programs, not Vault. The court then turned to the Prolok shrink-wrap license. First, the contract would be unenforceable as a "contract of adhesion" (overbearing contract), were it not for the Louisiana law favoring it. It was therefore necessary to decide whether the Louisiana law was valid. The court then held that the Louisiana law was not valid because it interfered with the operation of the federal copyright law. Accordingly, Vault was free to disassemble Prolok without liability despite the prohibitions in the shrink-wrap license.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 171 Medium 172 High 173

MicroStandards

Michael Smolin
Smolin & Associates
3428 Greer Road
Palo Alto, CA 94303

Future micro standards projects

As the needs of the professional community change, so too is there a changing need for standards. When old technology and its standards lapse, new needs become paramount. On this topic, MicroStandards presents some of the ideas being discussed in the Microprocessor Standards Committee (MSC) for new standards projects.

What's in the works

Last issue's MicroStandards contained a list of the 25 currently approved standards development projects under the Computer Society's Technical Committee for Microprocessors and Microcomputers. Also listed were the 14 IEEE standards adopted from drafts prepared under the TCMM. These standards span the range from backplanes to mnemonics, from floating-point arithmetic to operating system interfaces to information transfer. The existing projects also cover BIOS, communication buses, computer languages, an instrumentation bus, and a ruggedized backplane bus. Some of the projects were authorized this year; others date back five years. The TCMM makes a continuous effort to stay abreast of the need for standards in its areas of expertise.

What's coming up, near-term

The MSC, which oversees the TCMM's standards development projects, has established several study groups. These study groups are formed when the MSC or its sponsor, the TCMM, desires to determine the feasibility of developing a particular standard before committing to a standards development project. This testing of the waters may reveal that little support exists for such a standard, that the proposal would include technology beyond the current state of the art, or that some other impediment precludes its feasibility. Conversely, the study group may attract experts in the field to help refine the scope and parameters for a standard and then work toward developing a draft. Currently, the MSC oversees four study groups:

- the Architectural Study Group (higher levels of computer architecture),
- the Superbus Study Group (performance 10 times that of Futurebus),
- the CMOS Backplane Bus Study Group, and
- the Vehicle Bus Study Group (bus for vehicle peripherals, instrumentation, controls, and sensors).

These projects are likely to be approved in the near future and represent the next range of topics subject to standardization through TCMM/MSC efforts.

MSC plans to start another project soon for the standardization of slot-function IDs (proposed as a bus-independent extension to the related Macintosh II protocol). This protocol allows for switchless automatic system configuration.

Of course, as with other standards development projects, parties outside of the TCMM and outside of the IEEE will be solicited for their comments and participation. Articles and announcements usually appear in *IEEE Micro* and *Computer* magazines and elsewhere. A broad base of interested parties participating in and contributing to the development of consensus standards has always been the IEEE ideal. Readers who are interested in participating in new projects may write directly to me. They will then be contacted when their project of interest is initiated.

What's downstream?

At this point I appeal to *you*. Where do your interests and needs lie in the area of micro standards? In what subjects would you like standards developed to aid your work efforts? (I already know about faster buses; what else?) Remember that standards can be either reactive (making order from the chaos of numerous incompatible existing approaches) or proactive (anticipating a need before the world develops numerous incompatible solutions). Guides and recommended practices are also grist for the IEEE standards mill as they also come under the aegis of the IEEE Standards Board. (See October 1987 MicroStandards for discussion of the types of IEEE standards documents.)

Please send me your suggestions. (You may volunteer to help develop the draft, but it isn't required.) I even look forward to arguments for the position that certain areas might be ill-served by standards.

Of the 33 technical committees of the Computer Society of the IEEE, only 11 are involved in standards development. The TCMM sponsors about 35 percent of the standards development projects. I will forward suggestions outside of the TCMM's areas of expertise or interest to the pertinent technical committees.

Flash!

By the time you read this, the following sponsor ballots should be under way:

- P959, I/O Extension Bus (SBX),
- P970, Advanced Backplane Bus (Versabus),
- P1096, Multiplexed High-Performance Bus Structure (VSB), and
- Reaffirmation of *IEEE Standard 796, IEEE Standard for Microcomputer System Bus* (S-100).

If you are an interested party not on the sponsor balloting body (have not received a ballot) and would like to contribute comments, contact Louise Germani at the IEEE Standards Office, (212) 705-7960.

Flash, flash!!

The MSC has asked the TCMM to cancel its project P856, Methods for Evaluating Microprocessor Performance, for lack of progress over several years. If you are interested in the maintenance of this project, make yourself known to me or Louise Germani at the IEEE Standards Office. The MSC would like to continue the project if there are any interested volunteers.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 174 Medium 175 High 176

MicroReview

Continued from p. 5

cation of 417 percent is used to get a good idea of how the document will look on a 300-dot-per-inch laser printer, since the bitmapped graphics of the Macintosh screen use a density of 72 dots per inch (300/72 = 4.17). Furthermore, whatever the displayed magnification, you can select a mode in which the cursor is replaced by a magnifying glass icon. As this icon is moved over any part of the display, a small area around the cursor position is shown magnified by a factor of approximately 5.

Knuth intended TeX for the creation of beautiful books, especially books that contain a lot of mathematics. The 70 pages of *The TeXbook* that deal with producing mathematical formulas are the heart of TeX. The rest is an envelope of applied computer science worthy of much study and admiration. A macro facility allows mnemonically named and easy-to-use formatting instructions to be constructed from TeX primitives. A collection of these, called Plain TeX, is supplied with TeXtures and described in *The TeXbook*. Plain TeX allows you to achieve with simple formatting commands anything that you can produce with a WYSIWYG word processor like Word. TeX handles kerning, ligatures, varieties of spacing, and all of the niceties that have become standard in the production of fine books.

These typesetting subtleties are handled properly without your having to do anything. Also, TeX examines whole paragraphs to find optimal line breaks, and it examines pages to find the best page breaks, again automatically dealing with typesetting niceties like "widows" and "orphans."

A major benefit of TeX is portability. TeX implementations exist in a variety of computing environments, and TeX processes simple text files that can be produced anywhere and transported easily. For example, Unix tools can be used to generate TeX input files. These files can be processed by an implementation of TeX in the same Unix environment, or they can be sent over a serial line to a Macintosh, where they can be processed by TeXtures. This flexibility is unavailable with the "standard" Unix formatter, Troff.

As a tool for generating technical communications, TeXtures has several drawbacks. Like all formatters, it's slow, and the use of formatting commands is far less intuitive than the WYSIWYG user interface. Furthermore, the price is substantial, especially considering that TeX is free and all you're paying for is the Macintosh interface. Nonetheless, nothing else on the Macintosh will produce technical publications that look anywhere near as good. If you strive for perfection, then you'll want to consider TeXtures.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 177 Medium 178 High 179

Continued from p. 7

Copyright Office hears screen display testimony

Richard H. Stern

On September 9, 1987, the US Copyright Office held a public hearing to determine whether display screens should be registered separately from associated computer program code. Richard H. Stern submitted a written statement and testified on behalf of the Computer Society, together with Stephen R. Levine, vice chair, Technical Committee on Computer Graphics, and Alan Paller, president of AUI Data Graphics, a division of Computer Associates International, Inc.

In addition, witnesses appeared on behalf of Apple Computer, Lotus Development, and several trade associations.

Society position

The Computer Society's position agrees with a recent Atlanta federal court decision in the case of *Digital Communications Assoc., Inc. v. Softklone Distrib. Corp.* For copyright infringement purposes, the court held that screen displays should be considered apart from both the code used to generate them and the rest of the code in the associated computer program.

The society disagreed with the present Copyright Office practice of (a) identifying the screen display with the code of the computer program, (b) treating them as a single unit, and (c) refusing copyright registration of the screen separately from the code.

The society's witnesses based their position on the propositions that

- many codes can generate the same display,
- slight changes in a code can generate a very different display,
- the type of creativity used to devise menus and other screens is different from that evoked to write the code, and
- copyright registration of screens as such would give stronger protection to proprietors of computer programs, thereby encouraging investment and creativity in the field.

Computer Society resolution

The Board of Governors of the Computer Society of the IEEE approved the following resolution on September 8, 1987:

> "RESOLVED, that the Board of Governors...in order to encourage and promote creativity and investment in this aspect of the computer graphics field, recommends that the Copyright Office...should permit an owner of a computer program screen display to register the screen for copyright protection apart from the registration of the underlying code in the computer program that generates the display."

The Board also approved presentation of written comments on this issue as submitted to the Copyright Office by Richard H. Stern. Stern is chair of the intellectual property subcommittee of the Computer Society's Committee on Public Policy (COPP) and a member of the Editorial Board of *IEEE Micro*.

Testimony

The witnesses asserted that Copyright Office refusal to register screens in this way would cause business uncertainty and insecurity due to the unpredictable scope of copyright in computer programs. "Hydraulic pressure for protection of screens" would lead some courts to protect them anyway, if not separately as screen displays, then as an aspect of the "look and feel" of computer programs. But in adopting look and feel protection, the courts might well go too far and hinder the creativity of other designers of screens. This problem could occur if the courts unthinkingly extended copyright protections to utilitarian aspects of user interfaces, which should remain in the public domain.

The witnesses expressed some differences in viewpoint. Levine voiced concern that some courts may not appreciate how much manufacturers base some screen displays on their analysis of human factors. He felt that failure to recognize that fact could lock up this approach in copyright claims, to the detriment of software progress.

Paller argued that protecting screen displays by copyright would lead to much more social benefit than possible harm. He stated that there are many ways to design good user interfaces and that providing means of capital formation is the most pressing concern in the software industry.

At the hearing Apple Computer demonstrated new screen displays based on referencing laser video disks. Apple illustrated how the authorship of screen displays differs from that of computer program code. This difference, they contend, is evidence for separate registration of screen displays and related code.

In contrast, Lotus Development—now in litigation with alleged Lotus 1-2-3 cloners Mosaic Software and Paperback Software—urged the Copyright Office to continue its present practice of registering only the underlying computer program. Lotus contended that the concept of computer program (and, by the same token, the scope of its copyright registration) should be understood broadly and flexibly. This approach would sweep up many diverse things besides the literal lines of code. Single registration of all aspects of a computer program would thus provide a flexible mechanism for protecting software that would not be tied to its present nature or the techniques for writing it.

The Information Industry Association also supported the present Copyright Office practice of allowing a single registration for all aspects of a computer program. However, they indicated no opposition to optional registration of screens as such.

IBM filed a statement taking a similar position.

Jack Russo, a private sector witness, urged the Copyright Office to create a new kind of registration for computer programs. He proposed a unitary, combined form of copyright protection of (a) the code, (b) the screen displays, and (c) all other aspects of a computer program except for the hardware. This protection would include software specifications and user interface.

The Copyright Office plans to publish its conclusions after considering the statements made by the witnesses.

Current literature

If you need up-to-date product data and pricing of standard stocked industrial electronic components, look into the *Electronic Component Catalog* from Mouser Electronics. This free, 176-page catalog offers 16,000 items, including resistors, transformers, semiconductors, hardware, integrated circuits, and microprocessors.

Mouser Electronics, 2401 Highway 287 North, Mansfield, TX 76063; (800) 992-9943; in Texas, (817) 483-4422.

Informer Computer Terminals, Inc., offers the *Dial-Up Networking Primer* as a conceptual guide for both technical and nontechnical readers. This illustrated primer is a quick reference or a more complete data communications guide, depending upon need. The booklet discusses company products, dial-up networks, and user implementation requirements.

Informer Computer Terminals, Inc., 12781 Pala Drive, Garden Grove, CA 92641; (714) 891-1112; free upon request.

Howard W. Sams has recently published three computer-oriented tomes—one for would-be desktop publishers, another for Unix aficianados, and yet another for beginning/intermediate Mac programmers.

Personal Publishing with PC PageMaker reveals techniques for designing publications from one-page flyers to multipage journals. It provides hands-on instruction, numerous visual examples, and an explanation of typesetting terms. Topics include selecting the right hardware and software, assembling a personal publishing system, working with type, building master pages, and adding graphic elements.

Author Terry Ulick also wrote *Personal Publishing with the Macintosh* and founded several magazines on the subject.

Unix Communications is a reference guide for users who need to communicate with other users and other systems. The book follows on standard documentation with coverage of Unix mail, networking, and file transfer tools. Authors Bart Anderson, Bryan Costales, and Harry Henderson of the the Waite Group demonstrate use, control, and programming of Unix communication tools with examples.

MPW and Assembly Language Programming for the Macintosh is an introductory-level book on the Macintosh Programmer's Workshop that teaches the Macintosh assembly language. Programmers can write programs using the Macintosh Toolbox. Mouse events, windows, QuickDraw, and menus are covered. This tutorial includes the MPW shell command language and the 68000 Instruction Set with directives and toolbox traps. Author Scott Kronick is a core member of the Berkeley Macintosh Users Group.

Howard W. Sams & Company, 4300 West 62nd Street, Indianapolis, IN 46268; (317) 298-5400; 320 pp., $18.95; 350 pp., $26.95; and 352 pp., $24.95, respectively.

MicroTidbits

Now you can record directly over magneto-optic data, thanks to a new technique developed at **Carnegie Mellon's Magnetics Technology Center.** Before this discovery, magneto-optics users had to erase before they could rerecord. Mark Kryder et al. have eliminated the erasure step by using a temperature-change method to write directly over previous data.

AT&T and **DEC** researchers are developing test methods with the **National Bureau of Standards** for the IEEE Standard for Portable Operating System Environments, commonly known as Posix. Industry and users need test methods to ensure that new operating system environments conform to the proposed Posix standard. NBS is also developing a Federal Information Processing Standard based on Posix for federal government use.

Access Data Products has set up an electronic bulletin board entirely dedicated to PC connectivity and networking. The board operates 24 hours a day at no charge with dedicated phone lines at (914) 667-1841, (914) 667-1842, and (212) 319-7300.

A monthly newsletter for emulator users has hit the mails. *News & Notes* provides educational articles, new product information, product updates, and a special emulator clinic column. To obtain a free subscription, contact Tammie Adams, **Softaid, Inc.**, 8930 Route 108, Columbia, MD 21045; (301) 964-8455 or (800) 433-8812.

Brave new PCs projected

Can you believe that in five years a typical office system will have 10 times the computing power, 12 times the main memory, and nearly 100 times the mass storage capacity of current PCs?

The editors of the *Computer Industry Almanac* predict that a system comparable to today's $3000 model will boast a 32-bit microprocessor, 8M bytes of main memory, one 3.5-inch floppy-disk drive, and one hard-disk drive with 80M bytes of storage. They also envision a high-resolution color graphics display, a 9600-baud built-in modem, and a built-in LAN interface. The system printer could produce letter-quality, high-speed draft printing of text and graphics. This dream machine's performance could be as high as 3 MIPS.

A budget-model, $1000 system would be a scaled-down version of today's office PC with CD-ROM for recreational applications. It too would be fired by a 32-bit microprocessor but would harbor only 4M bytes of main memory. One 3.5-inch floppy-disk drive, a color display, one CD-ROM drive, and a letter-quality printer round out the possibilities. Its performance is estimated at 1 to 1.5 MIPS.

In addition to product forecasts, the 780-page *Computer Industry Almanac* includes an industry overview and rankings of 400 companies, 1000 key people, and 800 top products. Current data on advertising, salaries, and sales in the computer industry offer points of comparison.

Buy the almanac in book stores or from the publisher (add $2.00 for shipping): Computer Industry Almanac, Inc., 8111 LBJ Freeway, Dallas, TX 75251-1313; (214) 231-8735; hard cover $49.95, paperback $29.95.

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 183 Medium 184 High 185

New Products

Marlin H. Mickle
University of Pittsburgh

Send announcements of new microcomputer and microprocessor products, and products for review, to Managing Editor, IEEE Micro, *10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.*

Color images transmitted via telephone lines

Color Video Fax Corporation of America is marketing the Japanese Analogue & Digital Science system for transmission of still, color images. According to the company, its Eris system transmits 512 × 480-pixel × 8-bit images via standard telephone communication lines in less than one minute.

The Eris digital transmission contains automatic error correction, automatic sending and receiving capabilities, and remote control from personal computers for system expansion. Pictures can be sent over telephone lines with optional use of a television monitor, video camera, VHS recorder, laser disk, or computer. Pictures can be stored on a floppy disk and/or output in hard copy on Polaroid, Hitachi, and Sharp printers.

Contact the company for pricing.

Reader Service Number 1

Color Video Fax Corporation of America markets Japan's Eris color-image, digital transmission system.

Hypermedia organization comes to Apples

Apple Computer's HyperCard allows Macintosh users to organize, use, customize, and create new information with multiple information types such as text, graphics, video, music, voice, and animation. An English language-based scripting language allows users to write their own programs.

With HyperCard, users can organize and use information the way they think—by association, context, and hierarchy—while browsing and searching through large bodies of information. Users sort, make notes, type, or draw on these cards in the same way they might use paper index cards.

Upon activating a button, users can link a card or part of a card to another card or a stack of cards. Buttons also perform tasks such as dialing the telephone, sorting a stack, or finding a videodisc sequence.

HyperCard consists of a main disk, help disk, stack examples and ideas disk, and backup; it requires a 1M-byte RAM Macintosh with two 800k-byte floppy-disk drives.

Suggested retail price of Apple's three-disk HyperCard is $49. It will be included with all new Macintosh computers.

Reader Service Number 2

VLSI diagnostic workstation announced

The Schlumberger/ATE Integrated Diagnostic System IDS 5000 Workstation tests and diagnoses VLSI technology. The workstation combines scanning electron microscope technology with CAD/CAE tools.

With the IDS 5000, logic designers can observe behavior of a circuit while simultaneously monitoring circuit connectivity and predicted behavior through the schematic netlist, layout database, and simulation tools.

Delivery of the Schlumberger/ATE IDS 5000 is 60 days ARO. The list price is $495,000, including a one-year warranty.

Reader Service Number 3

DSP simulators run on IBM PCs

Software from Avocet Systems simulates and debugs Texas Instruments signal processing chips. AVSIM321 and 322 used on IBM PC compatibles work with the TMS 320 family of chips.

In operation the software produces registers, flags, and program and data memory on screen for manipulation by editing keys. Function keys control program flow for debugging. An Undo key uses the simulator's trace memory to back up one-at-a-time through recently executed instructions for error detection. Screen menus and a command mode are added features.

The Avocet Systems simulator/debuggers are priced at $379.

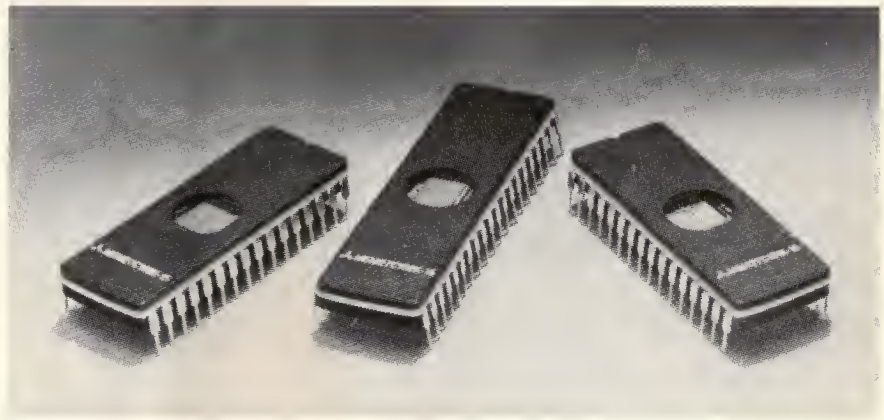
Reader Service Number 4

Mitsubishi offers three 150-ns, 1M-bit EPROMs

Mitsubishi Electronics America expanded its UV EPROM product line with three 150-ns, 1M-bit CMOS devices. Available in both 128K × 8- and 64K × 16-bit organizations, the EPROMs store nonvolatile memory in applications using 16/32-bit microprocessors.

Two of the devices, the M5M27C100K and M5M27C101K, are available in a 32-pin Cerdip package. The M5M27C-100K is pin compatible with the existing generation of 28-pin, 1M-bit mask ROMs. The M5M27C101K conforms to the JEDEC standard pinout for byte-wide memory.

A third device, the 64K × 16 M5M27-C102K, is encased in a 40-pin Cerdip package. All three offer a page programming algorithm that allows four bytes to be programmed at the same time.



Three CMOS EPROMs from Mitsubishi Electronics America.

Mitsubishi offers the devices for $40.75 each in 100-piece quantities for the 150-ns speed grade.

Reader Service Number 5

AT&T desktop computer uses 80386 chip

The 6386 WorkGroup System is part of AT&T's recently announced computer and data networking product line. The 6386 uses the Intel 80386 chip and includes a nonproprietary operating environment that lets it share software and data with the rest of the line and with other computers that use standard operating systems.

The 32-user system runs 32-bit Unix System V applications concurrently with MS-DOS applications and features a DOS Supervisor for running up to eight applications simultaneously at high speeds.

A 6386 desktop model includes 1M-byte RAM expandable to 48M bytes; a floor-standing model, the 6386E, includes 2M-byte RAM expandable to 64M bytes. Both models use either 5.25-inch floppy disks or 3.5-inch minifloppies.

List prices for the AT&T 6386 WorkGroup System begin at $4899.

Reader Service Number 6

CAE tools aid power circuit designs

Analog Design Tools has announced a set of design tools for the Analog Workbench that is focused on the simulation and analysis needs of designers working on power supplies and other power circuits.

The Power Design Tool Kit combines a nonlinear model of transformer core magnetics with libraries of magnetic core materials and semiconductor power devices. Users can select tolerance distribution functions for statistical analysis calculations. Air gap effects on core behavior can be modeled, and semiconductor thermal resistivities can be modified for stress analysis.

The Analog Design Tools software for the Analog Workbench with three libraries costs $33,000.

Reader Service Number 7

Network controls 1600 RS-232 devices

The CMCNETII communications network from Connecticut Micro-Computer enables an RS-232 computer or terminal to control a network of up to 1600 RS-232 devices at distances up to several miles. The network is recommended for factory automation and information gathering systems.

The system consists of four types of modules, a controller attached to the PC, device modules connected to each RS-232 device, expansion modules for systems of over 40 devices, and a repeater module for driving branches of the network more than 4000 feet.

CMCNETII modules cost under $100 in OEM quantities.

Reader Service Number 8

CMOS static RAMs feature 35-ns access times

Three CMOS static RAMs from Advanced Micro Devices are available in standard and low-power versions for commercial and military applications. Applications for the devices include cache and buffer memory, graphics, distributed processing, multiprocessing system, disk controllers, and instrumentation.

The 16K × 4-bit Am99C164 and Am99C165 operate at 605 mW maximum in standard-power versions and offer standby power at CMOS input levels of 82 mW. The low-power versions consume 495 mW operating power and 1.6 mW standby power at CMOS input levels. Their access time for military use is 45 ns.

The Am99C164 features chip enable and write enable controls for array application mode; the Am99C165 incorporates both functions and adds an output enable control, alleviating bus contention in high-speed systems with large memory banks.

The Am99C88H features access times of 35 ns for commercial use and 45 ns for the military version. The device is offered in standard and low-power versions. All three devices complement the Am99C641 64K × 1 static RAM.

In 100-piece quantities the DIP and leadless chip carrier devices are priced from $20 each.

Reader Service Number 9

Optical system design aid announced

Dynacomp has introduced three educational and scientific software packages for IBM PCs and compatibles. Optics One, an optical ray-tracing program, employs recursion and iteration schemes to minimize both truncation and round-off error. The menu-driven lens design software allows users to zoom in on a graphical display to fine-tune their designs. The positions of surfaces, locations and heights of images, and Petzval sums are included in the general report. Optics One requires a 256K RAM, MS-DOS 2.0 or higher, and a graphics card.

Plotsmith is another menu-driven package for data plotting up to 10 sets of 250 data points each. Users enter and edit data from the keyboard.

Eigen Analyzer, designed for Apple, CP/M, and IBM computers, helps users find the real and complex eigenvalues of a real matrix, whether specified uncertainties exist in the matrix elements or not. Users can enter, edit, and save elements for later recall from disk as ASCII files.

Dynacomp sells Optics One for $99.95 (disk plus manual) and Plotsmith and Eigen Analyzer for $49.95 each (disks).

Optics One **Reader Service Number 10**
Plotsmith **Reader Service Number 11**
Eigen Analyzer **Reader Service Number 12**

Software Engineers

Northrop Corporation's Defense Systems Division, located in sprawling Rolling Meadows, IL just northwest of Chicago, provides a state-of-the-art software development environment implemented on a VAX cluster configuration, running under VMS, connected to Sun work-stations on an Ethernet fiber optics LAN, running under UNIX. Each software engineer has a terminal with access to any system on the network. Terminals are being replaced by personal workstations.

We offer professionals with a BSCS, BSEE, BS Math or Physics (or equivalent) MS preferred, and a minimum of 3 years experience, opportunities in the following areas. **Management, Systems Architect, Technical Leaders and engineering assignments available.**

Systems Programmers

Our many, varied applications require significant growth in our support capabilities. We need the best people with experience in:

- LANGUAGES, including Ada, Assembler, C, FORTRAN, JOVIAL, and Pascal
- OPERATING SYSTEMS, including UNIX and VMS
- Development of Real-Time Operating Systems
- Development of Software Tools
- Performance Modeling and Evaluation
- Use of Structured Software Development Methodologies

Software Systems Engineers

Our software engineers develop software from systems requirements through implementation, and need experience in:

- Software Requirements Analysis
- Architectural Design
- Software Validation and Test Specification
- Performance Specification and Modeling
- Interface Design and Specification

ECM/EW Systems Software Engineers

ECM/EW Systems are our business. Experience should include:

- Real-Time Control Systems
- Radar Data Processing
- Embedded Computer Systems
- Systems and Unit Level Diagnostics
- Object Discrimination & Classification
- ECM Algorithm Development
- Kalman Filtering
- Optimal Control

Hardware Diagnostics Software Engineers

We design and develop advanced systems using the latest hardware and software technologies for our military clients. Experience required in:

- Intelligent Control Panel Systems Development
- Built-in-Test, Functional Test
- Micro and Macro Diagnostics for Fault Identification

Interested individuals are encouraged to forward resume to: **Supervisor-Staffing, Dept. C98, Northrop Corporation, Defense Systems Division, 600 Hicks Road, Rolling Meadows, IL 60008.** An equal opportunity employer M/F/V/H. U.S. Citizenship required.

NORTHROP
Defense Systems Division
Electronics Systems Group

PS/2 software links laptops, PCs

Meridian Technology's Carbon Copy Plus data communications program is available on 3.5-inch disks for use on IBM's PS/2 family and compatible portable laptop computers. The software combines PC-to-PC remote control, PC-to-host terminal emulation, and X-modem and Kermit file transfer protocols in an integrated package.

In remote-control mode, the program joins together two PCs over an asynchronous link so that a keystroke entered on one appears simultaneously on the other. In a second mode, Carbon Copy Plus emulates DEC VT-52 and VT-100, Televideo 920, and IBM 3101 asynchronous terminals. Users can access host computers and on-line information databases.

The Meridian Technology software costs $195.

Reader Service Number 13

VMEbus board supports TOP protocol

Motorola/Microcomputer Division's MVME374 controller board hosts an Ethernet chip set and supports the seven-layer Technical and Office Protocol with 32-bit data transfer.

The VMEbus-based board also contains 1M byte of 32-bit-wide shared DRAM with parity for one-wait-state access to RAM supporting the MC68020 MPU and zero-wait-state cycle access for Ethernet Lance chip use.

The accompanying MicroTOP 1.0 protocol software is based on current OSI software and patterned after the ITI-certified MicroMAP 2.1 protocol stack.

Motorola expects the MVME374 to reach production early in 1988 and sell for $1795. Software object codes should sell for $600 each and source codes for $50,000.

Board **Reader Service Number 14**
Software **Reader Service Number 15**

PS/2 tape systems store up to 600M bytes

Mountain Computer has expanded its line of tape backup and disk data storage with four products that support IBM Personal System/2 models.

Two tape systems operate under the DOS 3.3 operating system and include Advanced Version 4.4 tape backup software. Both are compatible with IBM PC Net, Token Ring, Novell, and Xenix System 5 SCO networks and store from 40M bytes to 600M bytes of data.

Two disk drive systems are available for immediate shipment. One is a 50M-byte version of DriveCard, which plugs into the PS/2 Model 30 expansion slot. The other is a dual-drive version of Mountain's MicroBernoulli. An external unit, it provides two removable, 20M-byte cartridges and operates at hard-disk speeds.

Tape unit prices begin at $595; disk drives, at $1095.

Tape units **Reader Service Number 16**
Disk drives **Reader Service Number 17**

Neurocomputer adds 80386 processing speed

The Anza AZ1500 neurocomputer system from Hecht-Nielsen Neurocomputer Corp. offers neural network researchers and application developers an 80386-based host computer. Typical preprocessing algorithms that benefit from the added speed include digitizing, normalization of data, and Fourier transforms. Applications include machine vision, speech recognition, and real-time signal analysis.

The host computer is a Zenith 386-80, which acts as an I/O device for the AZ1500 IBM and compatible neurocomputer coprocessor board. The neurocomputer implements virtual neural networks containing up to 30,000 processing elements with up to 480,000 interconnects. Included with the system is the HNC Neurosoft software of five Neural Net packages and User Interface Subroutine Library.

The Anza AZ1500, with neuro-computing coprocessor integrated into a Zenith 386-80 and a 13-inch color monitor, is available immediately for $19,500.

Reader Service Number 18

Lab, factory products offered for Macintoshes

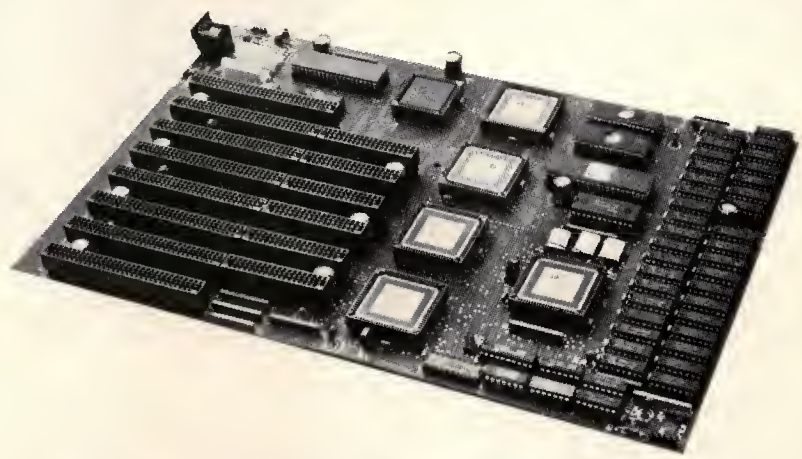
Six Macintosh II boards, two SE boards, and application software form the Analog Connection laboratory/factory series from Strawberry Tree Computers.

The Analog Connection series from Strawberry Tree Computers contains application software and eight plug-in boards to help users acquire and control data in laboratories and factories when using Apple's Macintosh II or SE computers. WorkBench application software ties together the boards, which provide 16-bit and 12-bit resolution and up to 16 analog input channels and 16 digital I/O lines.

Users select analog/digital inputs and outputs, data loggers, and chart recorders and tie them together on the screen to create symbolic representations of what actually happens in hardware. Typical use would include the measurement of voltage, temperature, pressure, strain, weight, and other parameters and control of heaters, boilers, pumps, valves, and other laboratory and industrial equipment.

Analog Connection WorkBench software costs $495; the boards begin in pricing at $595.

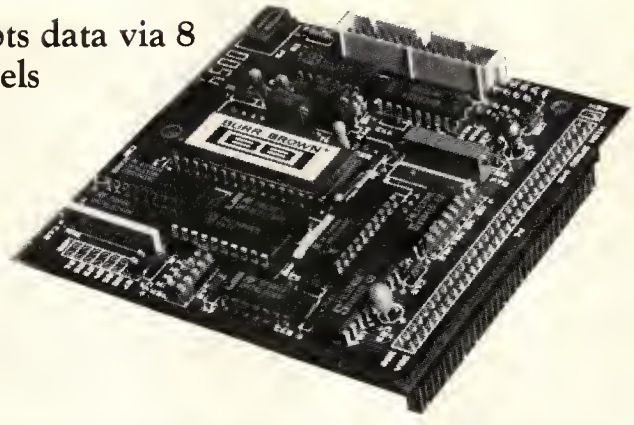
Software **Reader Service Number 19**
Boards **Reader Service Number 20**



VLSI Technology's VL82PCAT-SB1 lets users evaluate the features of the company's VL82CPCAT-QC chip set, which replaces PC/AT devices. The module contains a 12-MHz 80286 microprocessor and 36 120-ns, 256K × 1 dynamic RAMs for one-wait-state reads and writes at 12 MHz. $2000 each.

Reader Service Number 21

Module accepts data via 8 analog channels



Burr-Brown's Data Acquisition Module features a maximum multiplexer settling time of 3.5 μs, an A/D conversion time of 4 μs, an aperture jitter of 0.3 μs, and a total conversion time of 5.5 μs.

Burr-Brown has announced a 180K-channel-per-second data acquisition module designed to operate with the plug-in PCI-20000 Personal Computer Data Acquisition System.

The PCI-20023M-1 module provides eight single-ended analog-input channels and can be expanded to 128 channels. Intended for high-level signals, the module incorporates a sample/hold and A/D converter to provide high speed sampling rates. Internal hardware can configure the module to automatically increment to the next channel after each Start Convert command. Conversions can be started by the system's rate generator, an external signal, upon reading the previous conversion, or by software command.

Unit price of the PCI-20023M-1 is $895; delivery is from stock.

Reader Service Number 22

Unix coprocessor supports 32-bit computers

The Aeon:332/AT from Aeon Technologies is a coprocessor for the IBM AT and compatibles based on National Semiconductor's 32332 microprocessor. Designed for real-time and Unix applications, the board operates at 15 MHz with up to 4M bytes of no-wait-state DRAM. A 32382 demand-paged MMU allows full use of the 32-bit virtual address space.

Features include bus master capability, access to the AT-bus address space, and two 16-bit counter/timers. Operating system support includes Aeon5.3, Hunter and Ready's VRTX, and stand-alone access to DOS and BIOS functions. Languages available for Aeon5 on the 332/AT are C, Pascal, Fortran, Lisp, Basic, and Assembler.

Full-screen editors, symbolic source-level debuggers, and source code managers are included with the Aeon:332/AT.

The Aeon Technologies coprocessor is available with 1M byte of DRAM in OEM quantities 30 days ARO for $1995.

Reader Service Number 24

Digital sound converted to speech, music

The Covox Speech Thing is a digital sound converter that attaches to an IBM PC or compatible parallel printer port to produce speech, sound effects, and music for business, education, and pleasure.

Speech Thing includes an audio amplifier, built-in speaker and headphone jack, manual, and software for use as a resident talking appointment calendar, English/Spanish calculator, and game demonstration. Additional software includes a graphics-based sound editor, special-effects control panel, and prerecorded vocabularies for inclusion in user-written programs. Smooth Talker, the First Byte text-to-speech synthesizer, comes with Speech Thing; the Voice Master speech synthesizer and digitizer for voice recognition and stereo sound (used to prepare Speech Thing software) is available separately.

Covox sells Speech Thing for $69.95.

Reader Service Number 23

VME engine promises zero wait state

Heurikon Corporation's HK68/V2F features a VSB-compatible memory expansion bus, mailbox interrupt support, and one RS-232 serial port.

Heurikon Corporation has announced a 68020-based VME engine with 20-MHz, zero-wait-state operation for reads and one-wait-state for writes. Designed for real-time processing, the HK68/V2F offers up to 4M bytes of on-board DRAM with parity, 128K EPROM, 128 bytes of nonvolatile RAM, and optional 68881 floating-point coprocessor.

The HK68/V2F is available at 20-MHz operation for $1495 in OEM quantities of 100. A no-wait-state version is available for $2395 in the same quantities.

Reader Service Number 25

Handheld analyzer contains disk drive

Network Communications Corporation has developed a handheld network diagnostic instrument that includes a built-in, 3.5-inch floppy-disk drive. Its 50K to 700K data partitions, written in real time, can help users track down diagnostic problems by allowing them to compare before/after data or archive separate diagnostic tests. Users can also store programs, set up information on disk, and store or mail the minidisks.

Functions included in the 5-pound 6610D Network Probe are data line monitoring, protocol analysis, bit and block error-rate testing, RS-232C lead status analysis and visual display, and asynchronous terminal operation.

NCC sells the 6610D for $3800.

Reader Service Number 26

Atari ST scanner is menu driven

The ST Image Scanner from Navarone Industries inputs photographs, graphics, and technical drawings into the Atari ST computer via a Canon IX-12 interface.

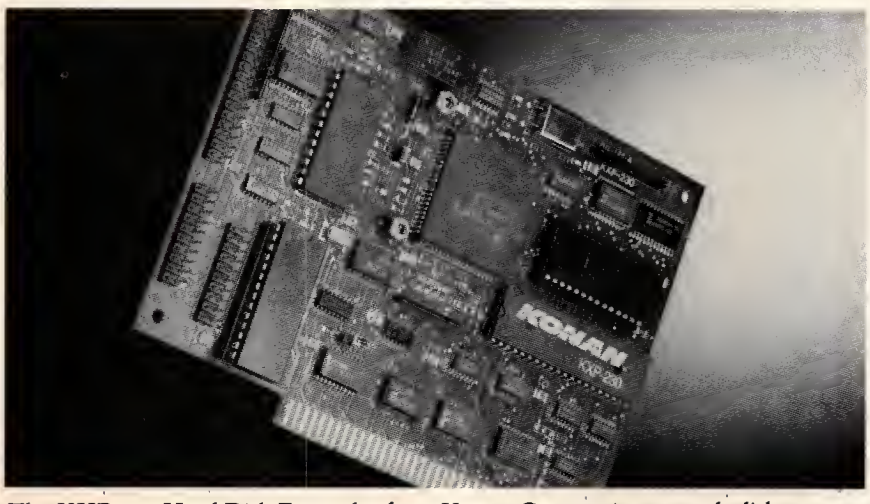
According to the manufacturer, the ST Image Scanner digitizes images in halftone mode for photographs or line-art mode for drawings and logos in less than 15 seconds. The entire image or selected area can be captured with resolutions of 75 to 300 dpi and 32 shades of gray.

The scanner software uses drop-down menu commands and is compatible with graphic programs that edit, crop, size, and print the image.

With interface, cable, and software, the Navarone Industries ST Image Scanner costs $1239.

Reader Service Number 27

PC hard disks expanded to 302M bytes



The KXP-230 Hard Disk Expander from Konan Corporation extends disk capacity and improves drive access and file loading times through data, directory, and FAT caching.

A hard-disk controller that Konan Corporation says will increase storage in IBM PC, Tandy, and compatible computers by 100 percent is now on the market.

Model KXP-230 Hard Disk Expander uses data compression and file compaction techniques to compress individual files of highly repetitive data by as much as 800 percent. Data is stored on the disk in normal Modified Frequency Modulation format, eliminating the need for special RLL-certified hard disks when the KXP-230 is used. The board supports hard disks or volumes up to 302M bytes.

KXP-230 contains a disk cache for storing often-used data, disk correction for errors up to 4096 bits in length, and fragmentation control for contiguous file storage.

Suggested retail pricing of Konan Corporation's KXP-230 is $249 for use on PC, XT, or Tandy 1000 computers and $299 for PC ATs.

Reader Service Number 28

CLASSIFIED ADS

RATES: $5.00 per line, $50 minimum charge (up to ten lines). Average six typeset words per line, nine lines per column inch. Add $4 for box number. Send copy at least one month prior to publication to: **Heidi Rex** or **Marian Tibayan, Classified Advertising, IEEE MICRO Magazine, 10661 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-8380.**

In order to conform to the Age Discrimination in Employment Act and to discourage age discrimination, *IEEE Micro* may reject any advertisement containing any of these phrases or similar ones: "...recent college grads...," "...1-4 years maximum experience...," "...up to 5 years experience...," or "...10 years maximum experience." *IEEE Micro* reserves the right to append to any advertisement, without specific notice to the advertiser, "Experience ranges are suggested minimum requirements, not maximums." *IEEE Micro* assumes that, since advertisers have been notified of this policy in advance, they agree that any experience requirements, whether stated as ranges or otherwise, will be construed by the reader as minimum requirements only.

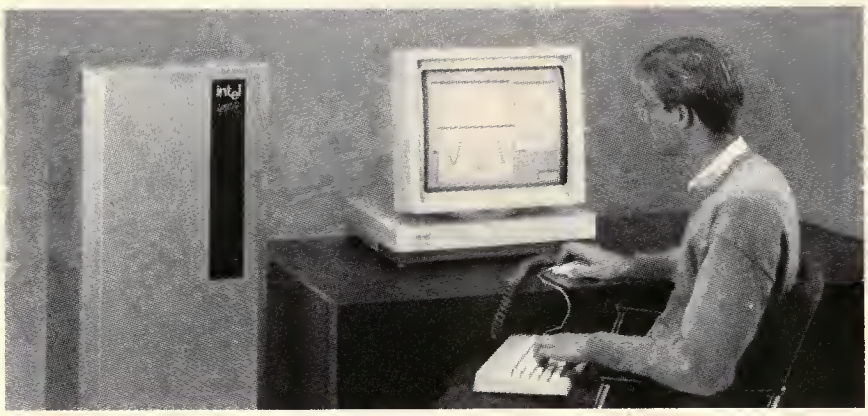
U.S. NAVAL ACADEMY
at Annapolis, Maryland

Seeking individuals qualifying for Assistant Professor position (tenure track).
Qualifications: An earned doctorate coupled with a strong background in the use of computing in the classroom.
Responsibilities: Assists faculty members with the integration of computer technology into the engineering curriculum. Must teach minimum of one undergraduate course per semester in the area of personal expertise.
Salary: Negotiable depending on experience and qualifications.
Personal expertise is expected to fall within one of the following Engineering disciplines: Aerospace, Electrical, Mechanical, Naval Architecture, Ocean, Marine, or Weapons & Systems.
Position is available June 1, 1988.
Closing date: December 31, 1987.
Submit letter of application, resume and minimum of three references to:
Director, Computer Services
Ward Hall
United States Naval Academy
Annapolis, Maryland 21402-5045
Attn: Mr. Doug Afdahl (301) 267-3693
(Equal Opportunity Employer).

WESTERN OREGON STATE COLLEGE
Computer Science Professor

Assistant professor to teach upper and lower division courses in operating systems, computer organization and data structures. Earned Doctorate in Computer Science required. A record of successful teaching experience preferred. 9-month, tenure-track appointment, $23,000 minimum salary. Send letter, curriculum vitae, and three letters of recommendation by February 1, 1988 to: Dr. David Olsen, Search Committee, WOSC, Monmouth, OR 97361, (503) 838-1220, ext. 509. Western is a liberal arts college of 3,650 students, which places special emphasis on undergraduate teaching. An Affirmative Action, Equal Opportunity Employer.

Second-generation hypercube offered



Intel's iPSC/2 system supports a fluid dynamics and heat transfer software package called Nekton from Nektonics, Inc. The numerical simulation tool solves complex 2D and 3D problems.

The Intel Scientific Computers second-generation hypercube features new hardware and software developments for easier programming and faster speeds. The iPSC/2 family of concurrent supercomputers can be used as compute servers for large-scale scientific and engineering applications.

Standard iPSC/2 systems are available in configurations from 16 to 128 processing nodes, with up to a gigabyte of memory. Vector concurrent iPSC/2 VX systems, with a vector arithmetic acceleration at each node and with up to 64 nodes, promise a peak performance of 400 MFLOPS.

Each system node features a 4-MIP 80386 and 80387. Surface-mounted 1M-bit DRAM modules provide 1 to 16M bytes of memory per node. Concurrent Workbench programming permits simultaneous access to the iPSC/2 system from a network of engineering workstations through a System Resource Manager interface, an 80386-based Unix V.3 system.

iPSC/2 pricing begins at $200,000. Original system upgrades are available.

Reader Service Number 29

Managers can forecast patent trends

Patents-PC software from Battelle allows corporate managers to track large quantities of data from the US Patent and World Patent Index databases. This data can be used to forecast technology developments, perform competitive analysis, and determine specific trends.

The software package includes a personal computer disk that can be copied onto a hard disk, sample database disk, user's guide and reference manual, and two hours of telephone consulting time with company experts. Intended for an IBM or compatible computer with a hard disk and 640K memory, the software is also compatible with Hercules, CGA, or EGA video monitors.

A site license for Patents-PC is available for $7500 for the first locale.

Reader Service Number 30

NS 32000's gain real-time operating system

Industrial Programming has announced a MTOS-UX real-time, multitasking operating system for National Semiconductor's 32000 series of microprocessors. According to the company, the MTOS-UX/32K operating system has the same internal structure and user inferface as other MTOS-UX computers, and programs written in a high-level language run in the same way.

A link to Unix permits a system or group of systems running under MTOS-UX to run as companions to a Unix-based workstation, in the manner of a work cell and its controller. It can log on and load object modules from files prepared under Unix.

Contact the company for pricing.

Reader Service Number 31

PC multiprocessor supports 64 users

Corollary Inc. has unveiled plans for a generic multiple processor system designed for 386 PC compatibles. Called the Attain 386, the system distributes user workload among up to five 386 processors, supporting up to 64 users.

Attain 386 includes an 80386 processor, 4M bytes of memory, and four specialized I/O ports that each interconnect to the company's current eight-line terminal concentrator. A 386 system can be configured with up to four Attain 386 processors, while maintaining SCO Xenix 386 binary compatibility for interchangeable, generic computing.

Corollary's plans call for December shipments; no pricing has yet been set.

Reader Service Number 32

Raster editor supports scanner

Houston Instrument's Hi-Scan Raster Graphics Toolkit enables users to edit and manipulate drawings scanned into a computer with the company's Scan-CAD plotter accessory. An icon-driven utility package, Hi-Scan works with up to E-size raster image files, allowing manual conversion to vector files or hard-copy output. Either a True Grid or Hipad digitizer, a mouse, or the keyboard arrow keys can be used to control the cursor.

Houston Instrument will mail Hi-Scan software to users who have purchased a Scan-CAD Model 128 and have applied for warranty registration.

Reader Service Number 33

Macintosh II drive accesses data in 26 ms

The newest member of the CMS Enhancements PRO-Series family of hard-disk subsystems is a 140M-byte internal drive designed for Apple Computer's Macintosh II.

Available immediately, the 5.25-inch, half-height PRO-140 II/i features an average access time of 26 ms. The subsystem includes the CMS SCSI Utilities program that assists in formatting, initializing, and installing the disk drive. Suggested retail price is $1695.

Reader Service Number 34

Digitize nondimensioned parts faster

PMX's Digi-Graph provides productivity improvements of up to 10 to 1 in digitizing complex, nondimensioned machine development parts and artwork. The software is useful for digitizing such geometries as die blank developments, artistic detailing, lettering, and logos.

With Digi-Graph users input straight-line moves, true arcs, and rapid motion statements. Lines are defined by their endpoints, and arcs by any three points on the circumference.

The IBM PC software creates DXF-formatted files for data exchange between XL/NC NC Programming software, AutoCad, Cadkey, and other DXF systems. The digitizing pads supported include GTCO, Pnumonics, Hewlett Packard, Kurta, Houston Instrument, and Skalar Systems.

Contact the company for price.

Reader Service Number 35

A/D converters offer 20-kHz sampling rates

Two 16-bit sampling A/D converters from Micro Networks can be used to digitize high-frequency signals in digital signal processing applications.

The MN6290 and MN6291 converters are constructed with internal, user-transparent, track-hold amplifiers. These amplifiers enable the units to sample and digitize dynamically changing input signals (with frequency content up to 10 kHz) at sampling rates up to 20 kHz.

MN6290 has a 10-volt input span; MN6291's span is 20 volts. Each features a 40-μ maximum conversion time, a 5-μ acquisition time, and a 10V/μ slew rate with precision of $\pm$ 0.001 percent linearity error. Both devices offer two electrical grades and two operating temperature ranges.

Pricing for each device begins at $252 for quantities up to 14; production quantities require 12 to 16 weeks for delivery.

Reader Service Number 36

Software analysis tools announced by Intel

Intel Corporation has introduced 80286 and 8086/8088 real-time software analysis tools that can be run on the IBM PC AT and XT. Called iPAT-286 and iPAT-86/88, these products allow software developers to monitor the execution of real- or protected-mode software for speed tuning and test analysis.

iPAT tools permit the host system to display high-level histograms, tables, and code coverage maps and examine the code generated by 8086/80286 compilers and assemblers. The analyzers use high-level-language symbolics (ASM, C, PL/M, Pascal, Ada, and Fortran) so designers can quantify code behavior at the module, procedure, statement, or address levels.

Intel's analyzers begin in price at $2995; both kits require the iPATCORE base system.

Reader Service Number 38

Laser system stores 210,000 pages of data

The MMi-100 optical system from Micro Mart Optisys comes with a Small Computer Systems Interface. It attaches to an IBM PC or compatible using a Host Adapter, which fits into an 8-bit PC slot.

OptiDriver from Micro Mart's Optisys Division allows IBM compatibles to use laser technology to store the equivalent of 1100 floppy disks on a 5.25-inch, reusable cartridge. Users access a WORM (write once, read many) optical disk drive as if it were a Winchester drive. The MMi-100 OptiDriver system can be used with MS-DOS and a variety of word processing programs.

OptiDriver sells for $49.95.

Reader Service Number 37

Color printer interfaces with mainframes

Mitsubishi International's Model CHC-635 nonimpact, thermal-transfer printer features 11-inch horizontal images for its B-size print output. The color printer can produce 270,000 colors for CAD/CAM/CAE hard-copy prints and transparencies.

CHC-635, when interfacing the Shinko Videoprocessor Model SPI 3-1, can be connected with IBM 5080, DEC, and other mainframe color systems. This interface features a five-second capture time. On demand, the video processor can multiplex up to four different terminals.

Contact the company for pricing.

Reader Service Number 39

Reader Interest Survey

Indicate your interest in this department by circling the appropriate number on the Reader Interest Card.

Low 180 Medium 181 High 182

Product Summary

Marlin H. Mickle
University of Pittsburgh

For more information, circle the appropriate Reader Service Number on the Reader Service Card at the back of the magazine.

Manufacturer	Model	Comments	Rs No.
Chips			
Advanced Micro Devices	Supernet chip set	Five-device, bipolar and CMOS chip set implements the ANSI 100M-bit/s FDDI network using fiber-optic media. Replacing six boards of discrete ICs, the LAN can be used for workstations, an interconnection of differing types of networks, and as a link between mainframes, minicomputers, personal computers, and peripherals in a distributed processing environment. $625 in 100-piece quantities.	60
Fujitsu Microelectronics	MB86950 Etherstar controller	LAN controller incorporates protocols for IEEE 802.3 CSMA/CD 10M-bit/s Ethernet and 1M-bit/s Starlan networks. Configurable for 8- or 16-bit bus interfaces, the 1.5-micron, CMOS chip links a host system to a LAN in cost-sensitive network applications connecting personal computers, workstations, disk drives, printers, and other devices. $18.50 in 5000 to 10,000 quantities; 88-pin plastic or ceramic, 84-pin plastic PLCC, or 80-pin flat-pack, surface-mount carrier packaging.	61
Intel Corporation	82385 32-bit controller	CHMOS III cache memory controller supports 20-MHz computers as one of four components in the company's 80386 microprocessor-based Computing Engine chip set. Features of the software-transparent chip include a "posted write" policy that eliminates wait states when writing to main memory and a bus-watching mechanism that ensures cache coherency without performance penalties. $125 each in quantities of 10,000.	62
Boards			
Raster Technologies	GX4000 graphics accelerators	Designed for use with Sun Microsystems workstations, ANSI PHIGS/PHIGS+ board sets plug into a VME backplane to yield 3D graphics for research and scientific 3D visualization, real-time simulation, C3I, geophysics, molecular modeling, and animation.	63
Thomas-Conrad Corporation	TC6046 ARC-Card/MC	Interface board allows users to network IBM PS/2 Micro Channel computers to the Arcnet token-passing LAN. Uses 16K bytes of memory address space, the IBM 16-bit data bus structure, and a dual-port, 2K data buffer. $549 each with two-year warranty.	64
Systems			
Gespac Inc.	Gescomp microcomputers	Real-time, multitasking systems are intended for use as process or cell controllers in factory automation applications, especially with embedded computers in tools and instrumentations. The top-of-the-line 8340-P/HF contains a 32-bit, 16.7-MHz 68020 microprocessor; 68881 arithmetic coprocessor; 2.5M-byte RAM; 1M-byte, 3.5-inch floppy disk; and 40M-byte hard disk. $3995 each, basic configuration; OEM discounts.	65

Manufacturer	Model	Comments	Rs No.
Northwest Instrument Systems	Software Analysis Workstation	National Semiconductor Series 32000 CPU combined with analysis software permits users to picture code behavior during real-time execution in the target system. The SAW system displays information in the symbolic terms of the user's program.	66
Vermont Microsystems	Cobra graphics processor	AutoCad processor draws images at 80,000 clipped vectors/s and offers 1024 × 800 resolution, 16 to 256 simultaneous colors from a 16.7-million palette, and single-screen operation. Optional ADI driver enhances AutoCAD with instantaneous pan and zoom capabilities and full-view inset for parallel real-time viewing. $2995 to $4195, depending on palette selection.	67
Ziltek Corporation	ICE-Engine/68K in-circuit emulator	Provides up to 12.5-MHz, real-time emulation for 68000-series microprocessors; can be connected with IBM PC and compatibles, other host computers, or a dumb terminal through RS-232C ports. Real-time trace memory consists of 63 bits by 4096 words. Up to six levels of nested breakpoints can be used. $5895 with manual and PC driver.	68
Peripherals			
Delkin Devices, USA	525 Extra PS/2 drive	External floppy for Personal System/2 series plugs into existing connector. The 5.25-inch drive allows IBM systems to read, write, and format standard 360K disks as the B drive. $325.	69
Floating Point Systems	P64/210 I/O subsystem	Parallel I/O subsystem for the FPS M64 series controls, collects, and processes data while working in parallel with the CPU. VMS-compatible subsystem can be used to configure the M64 series with multiple units, array processors, DEC computers, high-speed disks, high-density tapes, and high-speed A/D converters. Under $100,000.	70
Software			
Digitalk, Inc.	Smalltalk/V, Version 2.0	PC-based implementation of the Smalltalk object-oriented programming language supports 640 × 480 graphic modes of the IBM PS 2/25 and /30 computers. Bitmapped window enhancements (disk and code browsers, debuggers, free-drawing panes) display windows faster than earlier version. $99.95; upgrades, $25.	71

FREE FACTS. *FAST!*

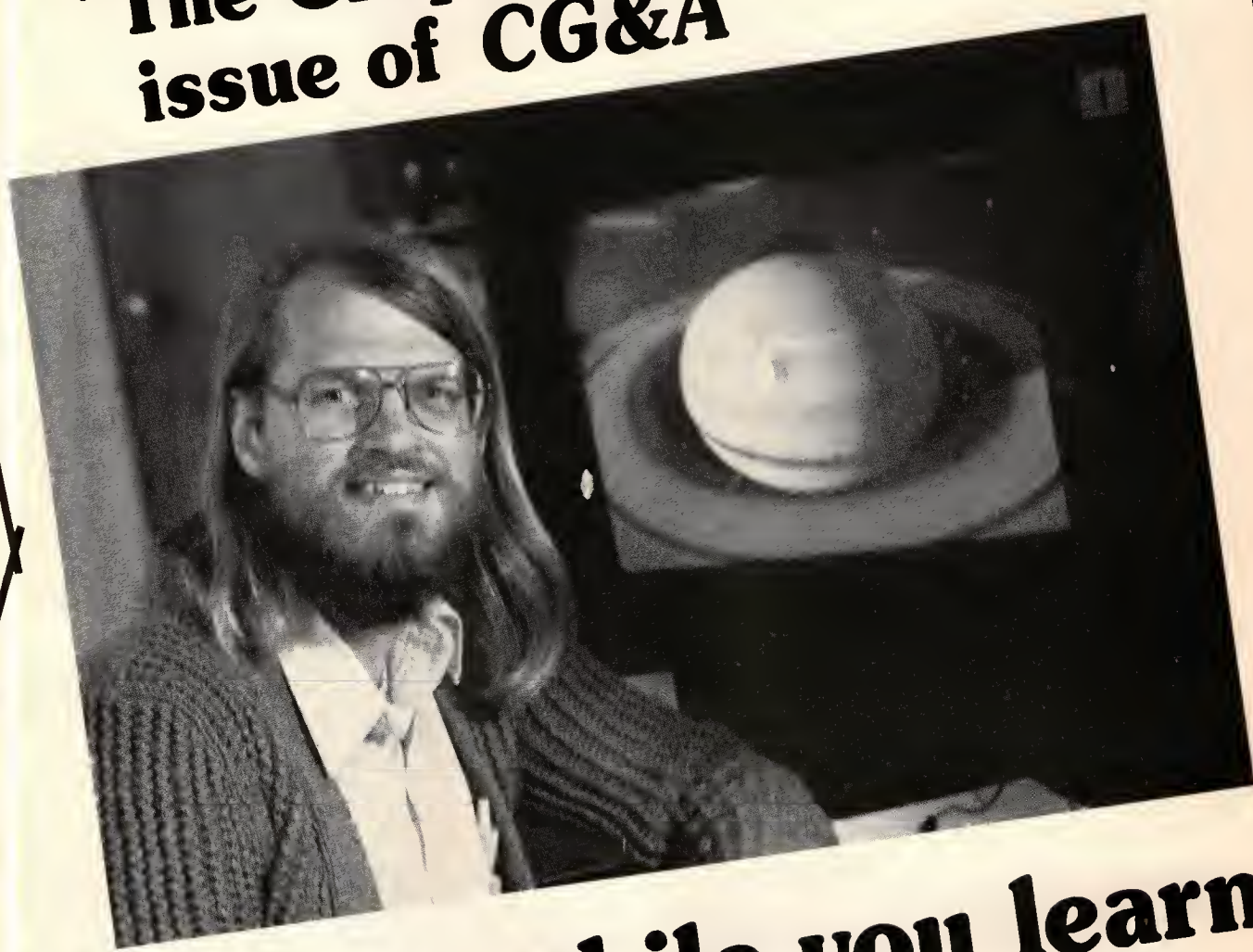
We've just sped up the response time to our Reader Service cards, so now you can get information more quickly about advertisers' products and services and the products we list in our New Products section. Under our new system, the company will have your name and address just days after you send out the card!

While you're indicating the products that interest you, please take a moment to circle the articles and departments that you liked in this issue. That will let us serve your interests better.

We'd like to hear from you!

Jim Blinn's Corner

Don't Miss "The Graphics Man" in every issue of CG&A



Laugh while you learn with the man who created the Mechanical Universe

COMPUTER GRAPHICS & APPLICATIONS
10662 Los Vaqueros Circle, Los Alamitos, CA 90720-9970
Just call (714) 821-8380

Calendar

Conferences sponsored or cosponsored by the Computer Society of the IEEE are indicated by the society's logo. Submit information **eight weeks before cover date** to Calendar, *IEEE Micro,* 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-2578.

February 1988

10th National Computer Conference (IEEE), Feb. 28-Mar. 2, Jeddah, Saudi Arabia. Contact Athar Javaid, IEEE, PO 8900, Jeddah 21492, Saudi Arabia.

Compcon Spring 88, Feb. 29-Mar. 4, San Francisco. Contact Hasan Alkhatib, Dept. of Electrical Engineering and Computer Science, Santa Clara University, Santa Clara, CA 95053; (408) 554-4485.

March 1988

1988 Spring National Design Engineering Show and Conference (materials and components), Mar. 7-10, Chicago, Illinois. Contact Show Manager, Spring National Design Engineering Show, 999 Summer St., Stamford, CT 06905; (203) 964-0000.

Southcon 88 (IEEE, ERA), Mar. 8-10, Orlando, Fla. Contact Southcon 88, 8110 Airport Blvd., Los Angeles, CA 90045; (213) 772-2965.

1988 International Zurich Seminar on Digital Communications, Mar. 8-11, Zurich. Contact Secretariat IZS 88, c/o P. Gunzburger, Hasler AG, TDS, Belpstrasse 23, CH-3000 Bern 14, Switzerland; phone 41 (31) 632-808.

Advances in Semiconductors and Superconductors: Physics and Device Applications, Mar. 13-18, Newport Beach, Calif. Contact SPIE, PO Box 10, Bellingham, WA 98227-0010; (206) 676-3290.

Seventh IEEE Phoenix Conference on Computers and Communications, Mar. 16-18, Scottsdale, Ariz. Contact Carl Ryan, Motorola GEG, 2501 S. Price Rd., Chandler, AZ 85248-2899; (602) 732-3074.

Compstan 88, Computer Standards Conference, Mar. 21-23, Arlington, Va. Contact Roger J. Martin, US Dept. of Commerce, National Bureau of Standards, Technology Bldg. 225, Rm. B266, Gaithersburg, MD 20899; (301) 975-3295.

Sixth IEEE VLSI Test Workshop, Mar. 22-23, Atlantic City, N.J. Contact Wesley E. Radcliffe, IBM East Fishkill, Dept. 277, Bldg. 321-5E1, Hopewell Junction, NY 12533; (914) 894-4346.

April 1988

Computer Networking Symposium, Apr. 11-13, Arlington, Va. Contact George K. Chang, Bell Communications and Research, 6 Corporation Pl., Piscataway, NJ 08854; (201) 699-3879.

CompEuro 88, Apr. 11-15, Brussels. Contact Jacques Tiberghien, VRIJE Universiteit Brussels, Pleinlaan 2, 1050 Brussels, Belgium; phone 32 (02) 641-2905.

Fourth International Conference on Metrology and Properties of Engineering Surfaces, Apr. 13-15, Gaithersburg, Md. Contact K. J. Stout, Coventry Polytechnic, Dept. of Mfg. Systems, Priory St., Coventry CV1 5FB, England; phone 0203 24166, ext. 278; or T.V. Vorburger, A117 Metrology Bldg., National Bureau of Standards, Gaithersburg, MD 20899; (301) 975-3493.

11th IEEE Workshop on Design for Testability, Apr. 19-22, Vail, Col. Contact T.W. Williams, IBM Corp., PO 1900, Dept. 67A/021, Boulder, CO 80301-9191; (303) 924-7692.

Workshop on Microstructure and Macromolecular Research with Cold Neutrons, Apr. 21-22, Gaithersburg, Md. Contact Charles J. Glinka, B106 Reactor Bldg., National Bureau of Standards, Gaithersburg, MD 20899; (301) 975-6242.

May 1988

19th Annual Pittsburgh Conference on Modeling and Simulation, May 5-6, Pittsburgh, Pa. Contact William G. Vogt or Marlin H. Mickle, Modeling and Simulation Conference, 348 Benedum Engineering Hall, University of Pittsburgh, Pittsburgh, PA 15261.

38th Electronic Components Conference (IEEE, EIA), May 9-11, Los Angeles. Contact Electronic Industries Assoc., 2001 Eye St. NW, Washington, DC 20006.

Fifth Workshop on Real-Time Operating Systems, May 12-13, Washington, DC. Contact John A. Stankovic, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA 01003; (413) 545-0720.

SIGMetrics Conference on Measurement and Modeling of Computer Systems (ACM), May 24-27, Santa Fe, N.M. Contact Connie U. Smith, L and S Computer Technology, 1114 Buckman Rd., Santa Fe, NM 87501; (505) 988-3811.

NCC 88, National Computer Conference (AFIPS, ACM, DPMA, SCS), May 31-June 3, Los Angeles. Contact AFIPS, 1899 Preston White Dr., Reston, VA 22091; (703) 620-8900.

June 1988

ISCAS 88, IEEE International Symposium on Circuits and Systems, June 7-9, Espoo, Finland. Contact Pekka Heinonen, Tampere University of Technology, Computer Systems Laboratory, PO 527, SF-33101 Tampere, Finland, or Olli Simula, Helsinki University of Technology, Dept. of Technical Physics, SF-02150 Espoo 15, Finland.

DAC 88, 25th ACM/IEEE Design Automation Conference, June 12-15, Anaheim, Calif. Contact Pat Pistilli, MP Associates, Inc., 7366 Old Mill Trail, Suite 102, Boulder, CO 80301; (303) 530-4333.

International Conference on Private Switching Systems and Networks (IEE), June 21-23, London. Contact Conference Services Dept., Institution of Electrical Engineers, Savoy Pl., London WC2R OBL, UK

18th International Symposium on Fault-Tolerant Computing, June 27-30, Tokyo. Contact Yasuo Komamiya, 2-4-8 Kikuna, Kohoku-ku, Yokohama 222, Japan; phone (81) 044-911-8181.

July 1988

Navy Micro/OA 88 Conference, July 25-28, San Diego. Contact Code 31.4, NARDAC San Diego, NAS North Island, Bldg. 1482, San Diego, CA 92135-5110; (619) 437-7013.

August 1988

Euromicro 88, 14th Symposium on Microprocessing and Microprogramming, Aug. 29-Sept. 1, Zurich. Contact Chiquita Snippe-Marlisa, PO Box 545, 7500 AM Enschede, The Netherlands; phone 31 (53) 338799.

September 1988

IEEE Artificial Neural Networks Conference, Sept. 18-21, Reston Va. Contact Kamal Karma, 823 Flegler Rd., Gaithersburg, MD 20879; (301) 984-7657.

Advertiser Index

FOR DISPLAY ADVERTISING INFORMATION CONTACT:

Northern California and Pacific Northwest: Don Farris Co., 161 W. 25th Ave., Ste. 102B, San Mateo, CA 94403; (415) 349-2222.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070.
Southern California and Mountain States: Richard C. Faust Co., 24050 Madison St., Ste. 100, Torrance, CA 90505; (213) 373-9604.
Southwest: The House Co., 3000 Weslayan, Ste. 345, Houston, TX 77027; (713) 622-2868.
Midwest: Robert Acker & Associates, 480 Central Ave., Northfield, IL 60093; (312) 441-6050.
East Coast: Hart Associates, PO Box 339, 42 Lake Blvd., Matawan, NJ 07747; (201) 583-8500.
New England: Arpin Associates, PO Box 227, Weston, MA 02193; (617) 899-5613.
Europe: Heinz J. Gorgens, Parkstrasse 8a, D-4054 Nettetal 1 - Hinsbeck (F.R.G); (0 21 53) 8 99 88.

Advertising Director: Dawn Peck.
For production information, conference, and classified advertising, contact Heidi Rex or Marian Tibayan.

IEEE Micro, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-8380.

Coming in the February issue of *IEEE Micro!*

- The structure and application of videorams
- An overview of R&D efforts in the area of GaAs microprocessors and digital systems design

Moving?

Name (Please Print)

New Address

City

State/Country Zip

☐ Address change: Please notify us 4 weeks in advance

Mail to:
IEEE Micro, Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-2578

ATTACH LABEL HERE

- This address change notice will apply to all IEEE publications to which you subscribe.
- List new address above.
- If you have a question about your subscription, place label here and clip this form to your letter.

Product Index

READER SERVICE CARD

IEEE MICRO

December 1987 issue
(card void after June 1988)

Name_____

Title_____

Company_____

Address_____

City_____ State___ ZIP_____

Country_____ Phone (___) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **1**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	191
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

READER SERVICE CARD

IEEE MICRO

December 1987 issue
(card void after June 1988)

Name_____

Title_____

Company_____

Address_____

City_____ State___ ZIP_____

Country_____ Phone (___) _____

Please send
(Circle those you want):

201 Publications catalog
202 Membership information
203 Student membership information
204 *IEEE Micro* subscription information

Reader interest
(Circle what you liked, add comments on the back)

Readers,
Indicate your interest in articles and departments **by circling the appropriate number** (shown on the last page of articles/departments) **in the shaded section of this card under Product Inquiries.**

Product inquiries **2**
(circle the numbers for products and advertisers you want more information on)

1	21	41	61	81	101	121	141	161	181
2	22	42	62	82	102	122	142	162	182
3	23	43	63	83	103	123	143	163	183
4	24	44	64	84	104	124	144	164	184
5	25	45	65	85	105	125	145	165	185
6	26	46	66	86	106	126	146	166	186
7	27	47	67	87	107	127	147	167	187
8	28	48	68	88	108	128	148	168	188
9	29	49	69	89	109	129	149	169	189
10	30	50	70	90	110	130	150	170	190
11	31	51	71	91	111	131	151	171	191
12	32	52	72	92	112	132	152	172	192
13	33	53	73	93	113	133	153	173	193
14	34	54	74	94	114	134	154	174	194
15	35	55	75	95	115	135	155	175	195
16	36	56	76	96	116	136	156	176	196
17	37	57	77	97	117	137	157	177	197
18	38	58	78	98	118	138	158	178	198
19	39	59	79	99	119	139	159	179	199
20	40	60	80	100	120	140	160	180	200

SUBSCRIPTION CARD

SUBSCRIBE TO IEEE MICRO

☐ YES, sign me up! ☐ Renew my subscription!

If you are a member of the Computer Society or any other IEEE society, pay the member rate of only **$17** for a year's subscription (6 issues). Subscriptions are annualized with membership. For orders submitted March through August, please pay half the given full-year fee for a half-year's subscription.

Society:_____ IEEE membership no:_____

☐ YES, sign me up! ☐ Renew my subscription!

If you are a member of ACM, NSPE, ACS, IEE (UK), SCS, IPSJ, IECEJ, or any other technical society, pay the sister society rate of only **$25** for a year's subscription (6 issues).

Society:_____ Mem. no. (if any):_____

_____
Full Signature *Date*

Name

Street

City

State/Country *ZIP/Postal Code*

☐ Payment enclosed .

☐ Charge to ☐ Visa ☐ MasterCard ☐ AmEx

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

Charge Card Number

Mo. Yr.
☐☐ ☐☐
Exp. Date

M1287

Charge orders also taken by phone:
(714) 821-8380 8:00 a.m. to 5:00 p.m. Pacific time
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720

Editorial comments

I liked: _____

_____

_____

_____

_____

I disliked: _____

_____

_____

I would like to see: _____

_____

_____

_____

_____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

Editorial comments

I liked: _____

_____

_____

_____

_____

I disliked: _____

_____

_____

I would like to see: _____

_____

_____

_____

_____

For reader service inquiries, use other side

PO box address for reader service cards only

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
PO Box 16508
North Hollywood, CA 91615-6508
USA

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 38 LOS ALAMITOS, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Computer Society of the IEEE
Circulation Dept.
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720-9970

THE COMPUTER SOCIETY

A member society of the Institute of Electrical and Electronics Engineers, Inc.

Executive Committee

President: Roy L. Russo*
IBM T.J. Watson Research Center
PO Box 218, Route 134
Yorktown Heights, NY 10598
(914) 945-3085

President-Elect: Edward A. Parrish, Jr.*

Vice Presidents
Education: Michael C. Mulder (1st VP)*
Technical Activities: Kenneth R. Anderson (2nd VP)*
Area Activities: Willis K. King†
Conferences and Tutorials: James H. Aylor†
Membership and Information: Merlin G. Smith†
Publications: J.T. Cain†
Standards: Helen M. Wood

Secretary: Duncan H. Lawrie
Treasurer: Joseph E. Urban
Past President: Martha Sloan*
Division V Director: Martha Sloan
Division VIII Director: H. Troy Nagle†
Executive Director: T. Michael Elliott†

*voting member of the Board of Governors
†nonvoting member of the Board of Governors

Board of Governors

Term Ending 1987	Term Ending 1988
Barry W. Boehm	Mario R. Barbacci
Paul L. Borrill	Victor R. Basili
Glen G. Langdon, Jr.	Lorraine M. Duvall
Duncan H. Lawrie	Michael Evangelist
Susan L. Rosenbaum	Allen L. Hankinson
Bruce D. Shriver	Laurel Kaleda
Harold S. Stone	Ted Lewis
Wing N. Toy	Ming T. Liu
Helen M. Wood	Earl E. Swartzlander, Jr.
Akihiko Yamada	Joseph E. Urban
Oscar N. Garcia†	

Next Board Meeting

8:30 a.m.-5 p.m., March 4, 1988
Cathedral Hill Hotel, San Francisco

Senior Staff

Executive Director: T. Michael Elliott
Editor and Publisher: True Seaborn
Director, Computer Society Press: Chip G. Stockton
Director, Conferences: William R. Habingreither
Director, Finance and Administration: Mary Ellen Curto

Offices of the Computer Society

Headquarters Office
1730 Massachusetts Ave. NW
Washington, DC 20036-1903
General Information: (202) 371-0101
Publications Orders: (800) 272-6657
Telex: 7108250437 IEEE COMPSO

Publications Office
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
Membership and General Information: (714) 821-8380

European Office
13, Avenue de l'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 770-21-98
Telex: 25387 AVVALB

Purpose

The Computer Society strives to advance the theory and practice of computer science and engineering. It promotes the exchange of technical information among its 90,000 members around the world, and provides a wide range of services which are available to both members and nonmembers.

Membership

Members receive the highly acclaimed monthly magazine *Computer*, discounts on all society publications, discounts to attend conferences, and opportunities to serve in various capacities. Membership is open to members, associate members, and student members of the IEEE, and to non-IEEE members who qualify as affiliate members of the Computer Society.

Publications

Periodicals. The society publishes six magazines (*Computer, IEEE Computer Graphics and Applications, IEEE Design & Test of Computers, IEEE Expert, IEEE Micro, IEEE Software*) and three research publications (*IEEE Transactions on Computers, IEEE Transactions on Pattern Analysis and Machine Intelligence, IEEE Transactions on Software Engineering*).

Conference Proceedings, Tutorial Texts, Standards Documents. The society publishes more than 100 new titles every year.

Computer. Received by all society members, *Computer* is an authoritative, easy-to-read monthly magazine containing tutorial, survey, and in-depth technical articles across the breadth of the computer field. Departments contain general and Computer Society news, conference coverage and calendar, interviews, new product and book reviews, etc.

All publications are available to members, nonmembers, libraries, and organizations.

Activities

Chapters. Over 100 regular and over 100 student chapters around the world provide the opportunity to interact with local colleagues, hear experts discuss technical issues, and serve the local professional community.

Technical Committees. Over 30 TCs provide the opportunity to interact with peers in technical specialty areas, receive newsletters, conduct conferences, tutorials, etc.

Standards Working Groups. Draft standards are written by over 60 SWGs in all areas of computer technology; after approval via vote, they become IEEE standards used throughout the industrial world.

Conferences/Educational Activities. The society holds about 100 conferences each year around the world and sponsors many educational activities, including computing sciences accreditation.

European Office

This office processes Computer Society membership applications and handles publication orders. Payments are accepted by cheques in Belgian francs, British pounds sterling, German marks, Swiss francs, or US dollars, or by American Express, Eurocard, MasterCard, or Visa credit cards.

Ombudsman

Members experiencing problems — late magazines, membership status problems, no answer to complaints — may write to the ombudsman at the Publications Office.

Information

Use the Reader Service Card to obtain the following material:

- Membership information and application (RS #202)
- Publications catalog (proceedings, tutorials, standards) (RS #201)
- Periodicals subscription application/information for individuals (members, sister-society members, others) (RS #200)
- Periodicals subscription application/information for organizations (libraries, companies, etc.) (RS #199)
- List of awards and award nomination forms (RS #198)
- Technical committee list and membership application (RS #197)
- Directory of officers, board members, committee chairs, representatives, staff, chapters, standards working groups, etc. (RS #196)

JOIN US.

THE COMPUTER SOCIETY OF THE IEEE

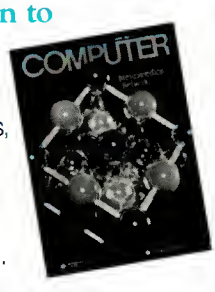
THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

The Computer Society gives you the balance you need.

From books on artificial intelligence to conferences on design automation, the Computer Society has the information you want and the balance between theory and practice you need.

We invite you to join us and receive these benefits:

An automatic subscription to COMPUTER Magazine



This authoritative yet readable monthly brings you industry surveys, tutorials, and application perspectives across the breadth of the computer field—from theory to practice, and every step in between.

Low member rates on other Computer Society magazines and transactions

Our magazines provide a practical application-oriented treatment of the leading edge of computer technology and our transactions provide the more analytical and theoretical base—the perfect balance between theory and practice.

The opportunity to work shoulder-to-shoulder with experts

in one or more of our standards working groups (we've got more than 80 now!) to develop draft standards, which after approval, become IEEE standards used throughout the world. (The Computer Society is unique in offering its members the opportunity to develop computer-related standards.)

Participate in one or more of our 33 technical committees—

networks of professionals with common interests in computer hardware, software, and applications. Technical committee programs are determined by the membership. Some are more theoretical; others are more applied.

Discounts on conference registration fees

Choose from more than 100 worldwide annually, from large applications-oriented conferences to small workshops, and a range of theory and practice for every interest.

To obtain membership information and an application, circle Reader Service Card #202.